Scheduling Garbage Collection in Embedded Systems

Roger Henriksson

CODEN: LUTEDX/(TECS-1008)/1-164/(1998)

Lund, July 1998

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund, Sweden

E-mail: Roger.Henriksson@dna.lth.se
WWW: http://www.dna.lth.se/~roger

Cover art by Ann-Marie Henriksson

© 1998 Roger Henriksson

Abstract

The complexity of systems for automatic control and other safety-critical applications grows rapidly. Computer software represents an increasing part of the complexity. As larger systems are developed, we need to find scalable techniques to manage the complexity in order to guarantee high product quality. Memory management is a key quality factor for these systems. Automatic memory management, or *garbage collection*, is a technique that significantly reduces the complex problem of correct memory management. The risk of software errors decreases and development time is reduced.

Garbage collection techniques suitable for interactive and soft real-time systems exist, but few approaches are suitable for systems with *hard* real-time requirements, such as control systems (embedded systems). One part of the problem is solved by incremental garbage collection algorithms, which have been presented before. We focus on *the scheduling problem* which forms the second part of the problem, i.e. how the work of a garbage collector should be scheduled in order to disturb the application program as little as possible. It is studied how a priori scheduling analysis of systems with automatic memory management can be made. The field of garbage collection research is thus joined with the field of scheduling analysis in order to produce a practical synthesis of the two fields.

A scheduling strategy is presented that employs the properties of control systems to ensure that no garbage collection work is performed during the execution of critical processes. The hard real-time part of the system is thus never disturbed by garbage collection work. Existing incremental garbage collection algorithms are adapted to the presented strategy. Necessary modifications of the algorithms and the real-time kernel are discussed. A standard scheduling analysis technique, rate monotonic analysis, is extended in order to make a priori analysis of the schedulability of the garbage collector possible.

The scheduling algorithm has been implemented in an industrially relevant real-time environment in order to show that the strategy is feasible in practice. The experimental evaluation shows that predictable behaviour and sub-millisecond worst-case delays can be achieved on standard hardware even by a non-optimized prototype garbage collector.

Acknowledgements

The research presented in this thesis was carried out within the Programming Environments Group at the Department of Computer Science, Lund University. I would like to thank my supervisor Boris Magnusson, the leader of the group, both for introducing me to the problem of scheduling garbage collection and for his support throughout my thesis work. Klas Nilsson, also a member of the group, deserves special thanks for his generous support and for sharing his knowledge of the automatic control domain with me. Thanks to Mats Bengtsson, a former member of the group, for introducing me to the secrets of real-time garbage collection.

The experimental work described in the thesis would not have been feasible without a fruitful cooperation with the staff at the Department of Automatic Control, Lund Institute of Technology. They provided both the physical means for the experiments and expertise in control systems. Special thanks go to Anders Blomdell for invaluable technical support, Anders Robertsson for the many hours he spent getting the inverted-pendulum experiment to run, and Johan Eker for his assistance regarding the Pålsjö system. Anders Ive, a member of the Programming Environment Group at the Department of Computer Science, also deserves thanks in this context for helping me to evaluate the performance of my prototype garbage collector. Thanks to the Department of Information Technology, Lund Institute of Technology, for generously making a logic analyser available to us during a long experimental phase.

Thanks to former and present members of the Programming Environments Group. It has been a pleasure to work with you all. Thank you Görel Hedin, Göran Fries, Ulf Asklund, Torsten Olsson, Elizabeth Bjarnason, Anders Dellien, Daniel Einarsson, Patrik Persson, and Mathias Haage. Thanks to Anne-Marie Westerberg for helping me to overcome all the academic red tape.

Finally, I would like to thank Ann-Marie Henriksson for the cover art.

This work has been financially supported by NUTEK, the Swedish National Board for Industrial and Technical Development.

Contents

Chapter 1 Introduction 1
1.1 Memory management 2
1.2 Real-time garbage collection 4
1.3 The thesis 5
1.4 Thesis outline 5

Chapter 2 Real-Time Systems 7
2.1 Real-time systems 7
2.1.1 Real-time requirements 8
2.1.2 Predictability 10
2.1.3 Control systems 11
2.2 Process scheduling 13
2.2.1 Static cyclic scheduling 13
2.2.2 Fixed priority dynamic scheduling 14
2.2.3 Earliest deadline first scheduling 18
2.3 Process scheduling in existing real-time kernels 19
2.4 Summary 20

Chapter 3 Automatic Memory Management 21
3.1 Introduction 21
3.2 Memory fragmentation 22
3.3 Basic garbage collection algorithms 23
3.3.1 Reference counting 23
3.3.2 Mark-Sweep 25
3.3.3 Copying algorithms 28
3.4 Conservative algorithms 32
3.5 Generation-based algorithms 33
3.6 Efficiency 34
3.7 Incremental algorithms 35

3.8 Scheduling properties 36
3.8.1 Stop-the-world 36
3.8.2 Interactive systems 37
3.8.3 Hard real-time computing 38
3.9 Memory hierarchies in real-time systems 39
3.10 Problem statement 41
3.11 Summary 42

Chapter 4 Scheduling Garbage Collection 43
4.1 Introduction 43
4.2 Semi-concurrent scheduling 44
4.3 Basic garbage collection algorithm 46
4.3.1 Tri-colour marking 46
4.3.2 Algorithm overview 47
4.3.3 The collector 48
4.3.4 The mutator 50
4.4 Scheduling the garbage collection work 51
4.4.1 Object initialization 51
4.4.2 Lazy evacuation 53
4.4.3 The high-priority garbage collection process 56
4.4.4 Distribution of GC work 57
4.4.5 Synchronization 60
4.5 Overhead 62
4.5.1 High-priority processes, service time 62
4.5.2 Low-priority processes, service time 63
4.5.3 Summary of worst-case mutator overhead 64
4.5.4 High-priority processes, latency 64
4.5.5 Cleaning up after the high-priority processes 65
4.5.6 Additional work for the programmer 65
4.6 Degradation during system overload 65
4.7 Measuring garbage collection work 68
4.7.1 Work metrics 68
4.7.2 The evacuation pointer metric 69
4.7.3 Improving the evacuation pointer metric 70
4.7.4 A fine-grained metric 71
4.7.5 Hardware support 72
4.7.6 Impact of imperfect metrics 72
4.7.7 Conclusions 72

4.8 Scheduling analysis 73
4.8.1 Schedulability of the high-priority processes 73
4.8.2 Schedulability of the garbage collector 76
4.8.3 Memory reserved for high-priority process usage 78
4.8.4 Scheduling analysis example 79
4.8.5 The effect of blocking 82
4.8.6 Priority inheritance protocols 82
4.9 Scheduling mark-sweep garbage collection 85
4.9.1 The algorithm 85
4.9.2 Atomic operations 89
4.9.3 Interruptible garbage collection 89
4.9.4 Work scheduling 92
4.10 Generation-based garbage collection 93
4.11 Summary 94

Chapter 5 A Garbage Collection Prototype 97
5.1 Environment 97
5.1.1 System architecture 97
5.1.2 Real-time kernel 98
5.2 The garbage collector 99
5.2.1 The algorithm 99
5.2.2 The garbage collector coroutine 99
5.2.3 Memory organization 101
5.2.4 Root pointer data structures 102
5.2.5 Real-time kernel modifications 103
5.2.6 Estimating garbage collection work 105
5.3 Application program interface 106
5.3.1 Initialization 106
5.3.2 Declaring objects 106
5.3.3 Pointer access 108
5.3.4 Pointer assignment 109
5.3.5 Allocation 109
5.3.6 Root pointers 109
5.3.7 Garbage collecting C++ objects 112
5.4 Discussion 114
5.5 Summary 115

Chapter 6 Experimental Results 117
6.1 Introduction 117
6.2 Experimental setup 118
6.3 Overview of experimental applications 120

6.4 Measurements of garbage collection costs 122
6.4.1 Pointer assignment 123
6.4.2 Memory allocation 124
6.4.3 Allocation cost of manual memory management 128
6.4.4 Latency for high-priority processes 129
6.4.5 Execution time for the garbage collector process ... 130
6.5 Using the garbage collector in control applications 131
6.5.1 Inverted pendulum control 131
6.5.2 Polynomial regulator 132
6.6 Summary 133

Chapter 7 Related Work 135
7.1 Incremental copying algorithms 135
7.1.1 Baker's algorithm 135
7.1.2 Brook's algorithm 136
7.1.3 The Appel-Ellis-Li collector 137
7.1.4 Real-time replication garbage collection 137
7.2 Non-moving garbage collection 137
7.2.1 The Treadmill 138
7.2.2 Yuasa's algorithm 138
7.3 Hardware-supported garbage collection 139
7.4 Concurrent garbage collection 141
7.5 Special treatment of high-priority processes 142
7.6 Summary 143

Chapter 8 Future Work 145
8.1 Implementation 145
8.2 Analysis 148

Chapter 9 Conclusions 151
9.1 Contributions 151
9.2 Consequences 153

Bibliography 155

Index 161

Chapter 1

Introduction

Computers are frequently used as integral parts of technical equipment, such as robots, medical apparatus, and aeroplanes. Such computer systems are often called embedded systems, since they can be viewed as being a part of, or embedded into, the piece of equipment in question. As the price and physical size of computer hardware have decreased, it has become increasingly attractive to use this new technology to implement more and more complex functionality.

A very important factor in the design and implementation of an embedded system is safety. The system must often be able to perform its task for very long periods without faults: days, months, or even years. The consequence of a break-down may in some cases be directly fatal, e.g. a respirator that stops working or an aeroplane crashing. Even if most embedded systems are not as vital as in these examples, a break-down often leads to high costs. For example, an interruption of the production might mean lost sales for a company. Most ordinary computer applications do not have any demands on safety that come close to those of embedded systems, even though it is vital that the output is correct. If a word processor breaks down from time to time, it is irritating and some work might be lost, but that is all. Safety requirements are thus especially high for embedded systems, because of the severe consequences of a failure.

The systems discussed in this thesis are real-time systems, i.e. systems for which not only the values of the output of the system determine success or fault, but also the time at which the output is produced matters. Practically all embedded systems belong to this category. Real-time systems are characterized by having to perform a set of tasks as responses to external stimuli. Each task has a deadline, before which the task must be completed. Correctness of a real-time system depends on its ability to meet its deadlines.

As the capacity of computer systems has grown so has the desire to incorporate more and more functionality into software. This is done both to make the hardware less complex and thus cheaper, and to increase the overall functionality of the sys-

tem. In turn, the complexity of the software grows and so do the resulting problems. Complex software is expensive to develop and maintain, and is also error-prone. One way of organizing large and complex software systems, object orientation, has been successfully used in many areas. Object orientation provides a powerful way of mapping concepts from the application domain to the application program in a well-structured manner [KM93]. The execution of the program is viewed as a simulation of the behaviour of a part of the physical world. The idea is that programs structured in an object-oriented manner are easier to understand and to maintain than software written using traditional techniques. Code reuse is also encouraged by powerful abstraction mechanisms. It could be expected that the construction of embedded systems would also benefit from using object oriented techniques. One major problem in doing so is that the powerful mechanisms of object orientation rely to a high degree on a dynamic execution environment and in particular on dynamic memory management. This traditionally conflicts with the demands on predictable execution times. The main motivation behind this thesis is to solve this problem.

1.1 Memory management

One very important issue affecting the safety of a real-time computer system is how memory is managed. In many of the current systems, *static memory management* is used. All the memory the real-time system needs is allocated at start-up. This means that all data structures must be allotted enough memory to satisfy the worst-case needs of the application. In some systems, e.g. real-time systems implemented using Pascal/D80 [ERS85], static memory management even includes reserving memory for individual procedure activation records, restricting how procedures may be called. For example, recursive procedure calls are not allowed in such a system. The advantage of such an approach is that the system will be highly predictable, an important property of a real-time system. The disadvantage is that it must be possible to calculate the maximum size of each data structure in advance. Enough physical memory must be available to simultaneously hold all data structures at their maximum size. Static memory management is obviously not very flexible and imposes undesired restrictions on the programmer. Much thought has to be devoted to designing the application. The programmer will often have to write extra code in order to circumvent the restrictions.

Dynamic memory management means that memory is allocated from a central memory pool as it is needed by the application. When an area of memory, an object, is not needed any longer, it is returned to the pool, making it available for other purposes. The application can more easily adapt itself to a changing environment and programmers are less restricted. Available memory is also used more effectively. Object-oriented languages rely to a large degree on dynamic memory allocation. Introducing object orientation in the development of real-time systems thus makes dynamic memory management highly important.

Manual memory management

Dynamic memory management is often implemented by letting the application manage the memory on its own, perhaps with some rudimentary support from the runtime system. Memory is explicitly deallocated by the application when it is not needed any longer. This is usually called *manual memory management*.

Manual memory management is, however, very error-prone. Two kinds of programming errors are very common when manual memory management is used, namely *dangling pointers* and *memory leaks*. Deallocating an object too early means that pointers remain to the object somewhere in the application that will later be dereferenced in the belief that the object is still present (a live object). Such dangling pointers typically cause the program to crash. Knowing when to release an object is a global problem; an object must not be deallocated until it can be guaranteed that the object will never be accessed again. Safe deallocation requires that we must know that no other part of the application has a pointer to the object, that will later be dereferenced. On the other hand, neglecting to deallocate objects when they are not needed any more means that the memory used by the objects will never be reused. Memory seems to "leak away" from the application, hence the name memory leak. Sooner or later, the application will run out of memory and crash. Since real-time systems tend to run for very long periods, running out of memory is inevitable in the presence of a memory leak, even if the leak is small. Memory management errors of these kinds are often very hard to find.

Manual memory management requires significant amounts of code just to manage the free store. This code must often be written from scratch whenever a new application is developed, which increases the complexity of the software and also the risk of programming errors being introduced. The globality of the deallocation problem means that the consequences of a small modification of a system can be difficult to foresee since it might require changes to arbitrary parts of the system.

Systems that do not compact the heap (the part of the memory used for holding dynamically allocated objects) as objects are allocated and deallocated often suffer from the problem of memory fragmentation. Systems for manual memory management typically belong to this class of systems. Fragmentation means that there are unused areas of memory between live objects, but the areas cannot be reused for new objects since they are not large enough. Fragmentation makes the amount of memory needed for the heap significantly larger and yields a higher cost for allocating new objects.

Altogether, manual memory management can in general not be considered suitable for real-time systems for safety reasons.

Automatic memory management

Many of the memory management problems (such as dangling pointers, memory leaks, fragmentation, and a large volume of memory management related code) are

avoided if *automatic memory management* is introduced. In this approach, the responsibility for identification and deallocation of dead objects, i.e. objects not used any longer, is delegated to the execution environment. The part of the runtime or operating system performing the task of finding and recycling the memory occupied by all "garbage" objects is called the garbage collector, and the process itself is called *garbage collection*, or *GC* for short. Most of the error-prone code which manages memory can be eliminated from the application, making it less complex and considerably safer. Automatic memory management thus appears to be very suitable for real-time systems. The difficulty with using garbage collection in real-time systems has so far been to guarantee short enough response times.

Static memory management leads to unnecessary software complexity and increased risks of errors. Manual, dynamic, memory management introduces software errors since the task of deciding when to deallocate objects is too complex. The conclusion is that in order to achieve safe embedded systems, automatic memory management must be used and the associated problems to guarantee short enough response times eliminated.

1.2 Real-time garbage collection

Garbage collection algorithms have been developed for a wide range of application types, but the techniques have, unfortunately, so far suffered from problems with complying with very strict real-time demands. They are in most cases targeted for batch or interactive systems and do not guarantee short enough response times.

Memory management inevitably involves some overhead. The overhead can consist of additional space requirements, additional time requirements, or more often a combination of the two. The time overhead is sometimes measured as the percentage of CPU time needed for memory management. For real-time systems this measure alone is not an adequate measure of the overhead. For such systems to meet their timing requirements, it is important that the garbage collector does not delay the application for extended periods. Operations that could cause the garbage collector to be invoked, thus interrupting the execution of the application program, must have a short and bounded time overhead.

When discussing overhead it is important to distinguish between the average-case overhead and the worst-case overhead. For most applications it is the average-case behaviour that is of interest, since that is what will typically be encountered. In real-time systems, we must guarantee that the timing requirements will be met in every possible situation. Therefore, it is the worst-case behaviour that must be studied. The correlation between good average-case performance and good worst-case performance for a memory management system can be very weak. In fact, improving the worst-case performance often means that the average-case performance is degraded, and vice versa.

It is important to remember that memory management overhead is not only present in systems with garbage collectors. Both time and space overhead are asso-

ciated with manual dynamic memory management as well as with static memory management. In such systems, the responsibility of memory management is transferred to the application itself, and the costs are very difficult to estimate. Since the amount of time and space overhead is easier to analyse and measure for systems using automatic memory management, it is tempting to draw the erroneous conclusion that garbage collection is always much more expensive than other memory management strategies. Both time and space overhead exist for other memory management strategies as well. Furthermore, the problem of memory fragmentation is also often ignored, which can represent a significant space overhead in the worst case [Rob71].

1.3 The thesis

The goal of the research presented in this thesis is to find methods for making automatic memory management feasible in embedded systems with very strict real-time demands, especially systems for automatic control.

The method used is to study existing approaches to automatic memory management for real-time systems and to try to adapt them to comply with the restrictions of hard real-time applications. The main interest is focused on how the work of the garbage collector should be scheduled in order to disturb the control program as little as possible. Little research has previously been devoted to this field.

Our approach is to develop a strategy for scheduling the GC work of traditional garbage collection algorithms such that it does not interfere with the part of the real-time system that has to meet tight deadlines. Different parts of the system have to meet varying demands on real-time performance. This property can be used to schedule the garbage collection work in the time slots where it will cause minimal disturbance. Using some knowledge of when and how the most time-critical parts of the system execute, the strategy can guarantee that garbage collection will never disturb these parts.

Parts of the work have previously been published in [Hen94], [MH95], [Hen96], and [Hen97].

1.4 Thesis outline

The remaining chapters are organized into three major parts: An overview of real-time systems and memory management techniques is given in the *background* chapters. A new approach to real-time garbage collection is presented in the *scheduling real-time garbage collection* chapters. The approach is evaluated and related work is presented in the *conclusions* chapters.

Background

- Chapter 2: Real-Time Systems
  Real-time systems are described with emphasis on embedded systems. Different execution models based on concurrent processes, and their associated scheduling strategies, are explained.
- Chapter 3: Automatic Memory Management
  Manual memory management is compared with automatic memory management. An overview of algorithms for garbage collection is given. The algorithms are classified according to their ability to meet real-time demands. It is explained why previously suggested scheduling strategies are unsuitable for systems with hard real-time requirements.

Scheduling real-time garbage collection

- Chapter 4: Scheduling Garbage Collection
  This chapter forms the core of the thesis. An approach to scheduling the work of a real-time garbage collector is presented. It is shown how the proposed scheduling strategy and garbage collection algorithm comply with hard real-time demands and how its worst-case performance can be analysed. Various implementation issues are discussed.
- Chapter 5: A Garbage Collection Prototype
  An implementation in C of the proposed garbage collector strategy is presented.
- Chapter 6: Experimental Results
  The prototype implementation described in Chapter 5 is evaluated. The cost of memory management is measured on running systems.

Conclusions

- Chapter 7: Related Work
  Some previous approaches to introducing garbage collection in real-time systems in general and hard real-time systems in special are surveyed. It is discussed how they relate to the work described in this thesis.
- Chapter 8: Future Work
  This chapter outlines possible areas of future research.
- Chapter 9: Conclusions
  The contributions of the work are summarized and discussed, and conclusions are drawn.

Chapter 2

Real-Time Systems

A presentation of real-time systems is necessary in order to put the work described in this thesis into the right context. This chapter provides descriptions of concepts in real-time computing which are used in the following chapters. Special emphasis is put on hard real-time systems. For a more in-depth survey of hard real-time systems, refer to [But97].

Real-time software is in most cases implemented by a number of cooperating *processes*. In this thesis we use the word process to denote a separate thread of execution running concurrently with other threads on the same processor [Dij68]. All processes share the same address space and a context switch (assigning the processor to another process) is a relatively cheap operation. Processes, using our terminology, are also known as threads, tasks, or light-weight processes. Observe the difference between our notion of processes and that of *operating system processes* which typically run in separate address spaces and have a large overhead for context switches.

2.1 Real-time systems

The task of a program is to transform a sequence of input, or stimuli, to a set of output data. For most software, correctness means that correct output is produced given a set of input. The time needed for the transformation has no influence on the correctness. A program needing a long time to produce the correct output will probably be considered inefficient, but it will, nevertheless, be correct.

In some situations, the *time* at which the output is produced is important for the correctness of the system. The software must meet different *deadlines* for various tasks. Systems that have this property are called *real-time systems* and the requirements concerning how long time the system may use, and how precise in time it must be, are called *real-time requirements* or *real-time demands*.

2.1.1 Real-time requirements

The importance of timely operation can be used to classify software. The rest of Section 2.1.1 describes such a classification system.

Batch systems

Many computer applications do not really have any real-time requirements at all. An example of such an application is a compiler. The input to programs of this kind is prepared and available before running the program. Output is calculated based on the input, after which the program terminates. Systems designed to support such applications are often called *batch systems*, since data is processed in batches. The correctness of the output is not affected by the time it takes to produce it. However, a certain amount of efficiency is of course desirable for the program to be useful in practice.

Interactive systems

Modern computer programs often interact with the user during the processing of data. The user submits a command to the program, which performs the desired action and presents the result. Then, a new command can be submitted. We say such systems are *interactive*. One example of such a system is a word processor. For the interaction to work, the time needed to perform a command must be reasonably short. Otherwise, the user will perceive the system as being sluggish and, if the delay is unexpected, perhaps misinterpret the state of operation. If, for example, a word processor every now and then would need several seconds to update the screen as a response to the user pressing a key, the user might get the impression that the word processor has missed the key press and repeat it. The result will obviously not be what was intended.

Interactive systems often have some degree of real-time demands, even if they are relatively relaxed. Response times of up to around half a second are often considered acceptable. Delays shorter than 0.1 seconds can in most cases not be noticed at all by a human user. An exception is systems performing on-screen animations, where shorter response times are required in order to achieve smooth movement. Failing to meet a deadline occasionally is, however, not very critical. In the word processor example, the user might delete the extra characters inserted by mistake. A missed deadline in the animation example causes a sudden jerk, which can often be tolerated if it is of rare occurrence.

Soft real-time systems

When the computer controls some kind of external equipment the real-time requirements tighten. Embedded computer systems often belong to this category. The response times required are typically shorter than for interactive systems, often somewhere in the range of 10 to 100 ms. To maintain control over the external equipment and to make it perform its task efficiently, it is important that the system is able to meet its deadlines. In soft real-time systems however, occasional failures to do so can be tolerated.

An example of a soft real-time system is a telephone exchange. In order to service the customers efficiently it must respond quickly to actions taken by the caller. When the caller lifts the receiver the exchange should immediately generate a dial tone and be ready to accept a telephone number. If the exchange fails to do so the caller might get the impression that the service is not available at the moment. Even though the system failed to service the individual caller, the integrity of the system as a whole was not affected.

The borderline between interactive systems and soft real-time systems is often difficult to draw. An system performing animations could, for example, very well be considered to be a soft real-time system due to its tight deadlines.

Hard real-time systems

Some real-time systems have processes that must meet very strict real-time demands. Failing to meet these very tight deadlines can lead to system failure. Many systems for automatic control belong to this category. Such systems are for example used for controlling the movements of a robot arm, steering aircrafts, and controlling the operation of machines in an industrial plant.

The current theory behind the algorithms used for automatic control assumes that the state of the external process being controlled is sampled at regular intervals, and that new control signals derived from the sample can be produced in a very short time or with a predictable delay [ÅW84]. The sampling frequencies also tend to be quite high, in the range of 100-1000 Hz, making efficiency even more important. The response times that must be guaranteed are thus very short, a fraction of the sampling period, i.e. well below 1 ms. Failing to meet deadlines may cause the control algorithms to be unstable.

It should be noted that many control systems will tolerate occasional missed deadlines without failure. The result will often only be suboptimal performance. Still, many safety-critical systems exist where hard real-time guarantees are required.

Actual real-time systems often have a mix of soft and hard real-time demands. A hard real-time system for automatic control will typically contain low-priority processes with soft deadlines as well as high-priority processes with hard deadlines. Low-priority processes can for example be used for user interaction.

2.1.2 Predictability

An important property of real-time systems, especially hard real-time systems, is predictability. When we say that a system is predictable, we mean that upper (and sometimes also lower) bounds on the worst-case response times of the system exist and that it is possible to compute them. There are no delays of arbitrary length in a predictable system. In order to guarantee a maximum response time, each primitive operation performed by the system must be predictable. Predictability is the property that makes it possible to guarantee that no deadlines are violated.

Average-case versus worst-case performance

When evaluating the performance of a system, we study the time required to perform different tasks. One can either concentrate on how the system performs in the *average* case or in the *worst* case. If the average-case performance is to be studied, the mean execution time is determined, either by measurements or by a theoretical analysis of the code. The worst-case performance, on the other hand, is found by a similar analysis of the longest possible required execution time.

For batch and interactive systems we are usually only concerned with the average-case behaviour in order to obtain acceptable performance. The system might once in a while fail to meet a deadline, but this is of minor importance.

For real-time systems, especially those with hard real-time requirements, it must be possible to guarantee that the systems never fail to meet a deadline, at least not a hard one. Good average-case performance is desirable, but the ability to meet the deadlines in a worst-case situation is imperative.

Verifying schedulability

There are basically two approaches for the developer of a safety-critical real-time system to verify that a system will meet its deadlines. One method is to actually run the software and measure the performance of the system. There are two major drawbacks to this approach. First, it is very difficult to ensure that the measurements capture the worst-case execution situation. Therefore, the measurements will in most cases be inconclusive. Second, it is often not possible to test the software in the actual application environment since failures to meet the deadlines will have too severe consequences. One cannot trust the software to run without having verified it, but one can not verify it without running it.

The second approach to verifying the schedulability of a real-time system is to do a theoretical analysis of the software processes before actually running the system. This is called *a priori schedulability analysis*. Using knowledge about the processes of the system, it is possible to analyse the timing when the processes are executed. The worst-case execution situations are studied and the developer

checks that all critical processes will meet their deadlines. The drawback of this method is that it requires intimate knowledge about both the hardware and the software. Execution pattern, deadline, and worst-case execution time (*WCET* for short) must be known for each process. Furthermore, possible blocking caused by communication between processes and access to shared resources must be taken into account.

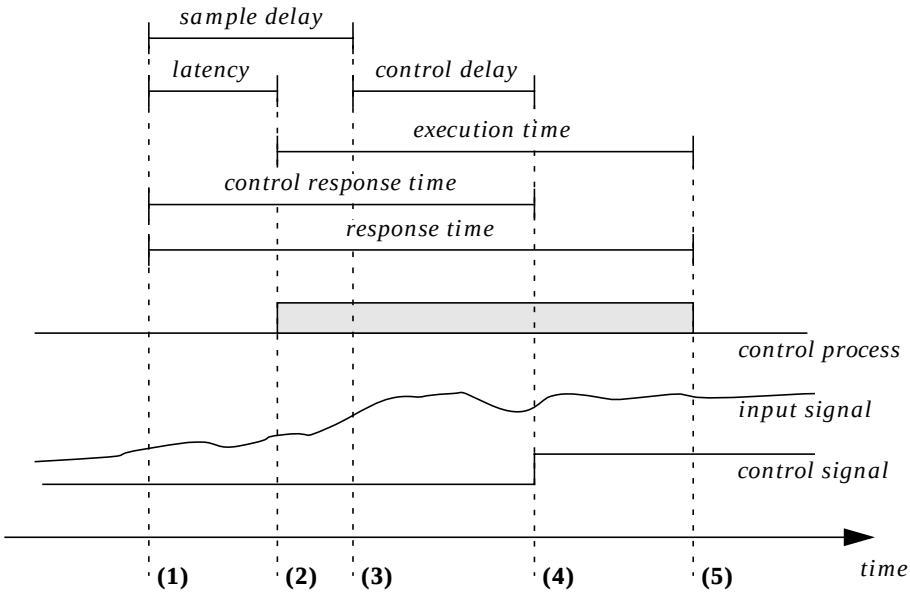
A conservative estimate of the WCET is acceptable from a schedulability analysis point of view. As long as the estimated value is not smaller than the real one, the scheduling analysis can still be trusted. If the analysis claims that a system is schedulable, it will also be schedulable in practice. However, if we use very conservative estimations, the analysis will classify many actually schedulable systems as unschedulable. Therefore, we want the estimated WCETs to match the real WCETs as well as possible.

Deriving close, but still conservative, estimates of WCETs is a non-trivial problem. One way of doing this is to study the code generated by the compiler for each process. The maximum execution time for each individual instruction is added together. Branches and loops complicate the analysis and some extra information is typically required from the programmer, e.g. the maximum number of possible iterations for each loop. The effects of pipelines and caches further complicate the analysis. The estimates can be calculated manually or produced by running the code through a simulator, e.g. [ATT88]. Another way to estimate the WCET of a process is to actually run the code of the process on the target hardware with a variety of input and measure the execution time. The maximum encountered execution time is then said to be the WCET. There are of course no guarantees at all that the worst possible case really did occur while the measurements were performed. To compensate for this to some degree, one usually adds some extra time to the measured worst case, perhaps 10%. Even though this approach does not produce any result that can be absolutely trusted, it is probably one of the most used method in practice.

2.1.3 Control systems

Control software is normally organized as a number of periodically executing processes. Each period, these processes sample the inputs, calculate new control signals and output. Because the established control theory of today demands that the samples are taken periodically, and because the time between sampling and outputting new control signals should be short in order to obtain optimal performance, these processes are assigned high priorities. High-priority processes are usually small and run only a short period of time each time they are activated. The sampling frequency required by the control algorithm varies, but for hard real-time systems, e.g. motion control, it is often in the range of 100-1000 Hz.

Figure 2.1 shows one invocation of a typical high-priority control process. When it is time to take a new sample and update the control signals (at the ideal sample



1. The control process is *released* (the ideal sampling time).
2. The control process is *invoked*.
3. Actual sampling time.
4. New control signals are output.
5. The control process has finished its work.

Figure 2.1 Execution and timing nomenclature for a high-priority sampling control process.

time), the real-time kernel *releases* the control process. This means that the process is made *ready*, but it is not necessarily allowed to execute immediately. The time it takes from releasing the process until it is *invoked*, i.e. actually starts executing, is called the *latency* of the process. Typical control systems require latencies in the range of 1 ms or less. One of the first things the process does is to sample the state of the controlled environment. The time from the ideal sampling time until the sampling is actually performed is called the *sample delay*. The *control delay* is the time required to analyse the sample, calculate new control signals, and making them available to the actuators. Control theory requires that the control delay is kept short, in many cases below 1 ms, in order to guarantee stability and good performance of the control system. It is also desirable from a control theory point of view that the *control response time* is kept short. This should not be confused with the *response time* of the process, which is defined as the time from the process is released until it has finished executing. The latter definition is used when discussing the schedulability of the process.

Apart from a relative small number of really critical high-priority processes, embedded control systems contain a number of processes with *low priority*, which usually represent the major part of the system (at least in terms of code size). These processes typically do things like computing reference values, presenting the state of the controlled process to an operator and accepting operator commands, communicating with other computers, etc. The low-priority processes must also satisfy some real-time requirements, but the demands are closer to those of interactive or soft real-time systems than to those of hard real-time systems. The consequences of a missed deadline are also much less severe than for the high-priority processes.

To summarize, an embedded control system is often implemented using a number of concurrently executing processes. The critical tasks of the system are isolated to a small number of high priority processes with very tight real-time demands. It is of vital importance that these processes are able to run on time and without interruption. At the same time, the system contains a number of low-priority processes with relaxed real-time demands. The requirements of these processes are similar to those found in interactive or soft real-time systems.

2.2 Process scheduling

In order to design a working memory management strategy, it is important to be familiar with the environment in which it will be working. As mentioned earlier, embedded systems are typically implemented as a set of concurrently executing software processes. Since the number of software processes is typically much larger than the number of available processors, the processor time must be shared by the different software processes. Many embedded systems consist of only one processor. A special piece of system software, called the *process scheduler*, is responsible for scheduling the work of the processes. The process scheduler can employ various strategies for dividing the available processor time among the software processes. In the rest of this section we will present some of the most common scheduling strategies in short together with their properties.

2.2.1 Static cyclic scheduling

One of the oldest scheduling strategies is static cyclic scheduling. It assumes that all tasks are implemented by periodically executing processes. The processor time is divided into time slots and a static scheduling table is constructed when the system is designed. Each entry in the scheduling table corresponds to a time slot and determines which software process should be invoked at the start of that slot. If the process does not utilize the entire time slot, the system will be idle for the rest of the slot, wasting processor time. On the other hand, the process must make sure to finish before the time slot ends. The scheduler traverses the table, starting the

processes one by one. When the last entry in the table has been processed, the execution restarts at the beginning of the table. In some cases, the otherwise wasted part of each time slot is used to execute background tasks. This yields a more effective use of the available CPU time.

The strategy has the advantages that it is very predictable and easy to analyse. It is sufficient to make sure that each process occurs often enough in the table to meet its deadline and that the WCET of each process will fit within the associated time slot. It might be necessary to split a process with a long WCET into several shorter ones in order to make it fit within the time slots. Building the scheduling table can be somewhat problematic, since the problem is generally NP-hard. A very commonly used, but time-consuming, approach is to build the table by hand. Modifying the software of the system can be quite costly. If a new process has to be added, or the WCET of a process changes, a new scheduling table must be constructed from scratch.

Since the scheduler repeatedly traverses the scheduling table we are limited to using periodically executing processes. Sporadic events must be handled by polling for the events. If a very short response time is required for such an events, the time slots must be very short and a lot of time slots must be allocated for handling the event. This can be very wasteful.

The length of the scheduling table is determined by the periods of the processes that is to be scheduled. The scheduling pattern of a system of processes executing with fixed periods will repeat itself at fixed intervals. In order to avoid anomalies in the execution periods when the scheduler moves from the last entry in the scheduling table to the first one, the length of the table must correspond to the repetition interval of the processes. The shortest repetition length of a set of processes is the least common multiple of their execution periods, which can be large if special care is not taken when assigning periods to the processes. To avoid excessive table length one often manipulates the periods, i.e. makes them shorter, so that they are multiples of each other. However, this also means that more processor time is required.

2.2.2 Fixed priority dynamic scheduling

By *dynamic scheduling* we mean that the process scheduler dynamically decides which process to assign processor time to as the system is running. A static scheduling table does not exist. Processes are invoked only if they have work to perform, that is when they are *ready*. Sporadic events might cause a process to become ready, which eliminates the need for polling for such events. Processes can be periodic or sporadic and process periods do not have to be harmonized. Dynamic scheduling thus provides a more flexible environment for the software developer than static cyclic scheduling.

The criteria the scheduler uses to decide which process to run varies. A popular method is to assign unique priorities to the different processes. It is assumed, from

reasons of a priori analysis, that the priorities remain fixed during runtime. At any point in time, the processor is assigned to the process with highest priority among the set of ready processes. If a process with higher priority than the currently executing one becomes ready, the scheduler suspends, or *preempts*, the currently executing process and assigns the processor to the process with higher priority. This scheduling strategy is often referred to as *fixed priority scheduling with preemption*.

Various methods exist for assigning priorities to processes. The simplest ones are based on heuristics. One might for example order the processes according to their "importance" and assign priorities accordingly. However, there are no guarantees that such methods will produce the best possible configuration of priorities. It might even be that the resulting system is unable to meet some of its deadlines, while another assignment of priorities would have produced a schedulable system. In order to find an optimal solution to the priority assignment problem, we must first construct a model of the system, which we then can analyse. The models will by necessity be simplifications of the real world and impose some restrictions on how we design our software. Models complex enough to handle most practical real-time applications have, however, been developed.

A property of fixed priority scheduling is that it is not generally possible to achieve a 100% processor utilization ratio without missing deadlines. On the other hand, overload is handled in a reasonable way. When the system is overloaded, the preemptive scheduler still makes sure that enough processor time is given to the processes with high priority. In most cases, these are also the most critical processes. The most important tasks of the system are thus still performed on time. It is the processes with low priority, which typically only have soft real time demands anyway, that will miss their deadlines first.

Rate monotonic scheduling

Rate monotonic scheduling, *RMS* for short, is a variant of fixed priority scheduling where priorities are set monotonically according to the rate, or period, of the processes. That is, the process with shortest period should be assigned the highest priority and so on. The basic model assumes that all processes are periodic and have a fixed period. It is further assumed that the deadline of every process is equal to its period. Processes are not allowed to block each other or suspend themselves, except to wait for the next period to start. For such a system, Liu and Layland have shown that RMS is optimal [LL73]. If RMS does not produce a schedulable process set, nor will any other fixed priority scheduling strategy.

Rate monotonic analysis, *RMA*, provides a framework for analysing the schedulability of a process set scheduled according to RMS. The earliest schedulability test, that of Liu and Layland, provides a sufficient, but not necessary, condition for a process set to be schedulable. This means that the analysis might turn out to be inconclusive. A process set may be schedulable even if it does not pass the schedu-

lability test. Later, Joseph and Pandya presented an exact analysis for RMS [JP86].

The restrictions imposed on the processes by RMS can be very unpractical in a real-world situation. For example, practically every real-time system requires that processes communicate with each other. Communication between processes unavoidably gives rise to critical sections that may cause blocking to occur. Another example of an unpractical restriction is to demand that every process is strictly periodic. Generalizing RMA to handle realistic systems better has therefore been an active research area. Techniques have been developed to incorporate sporadic processes, process blocking, deadlines shorter then the process period, scheduling overhead, release jitter, etc. into the analysis [SRL94].

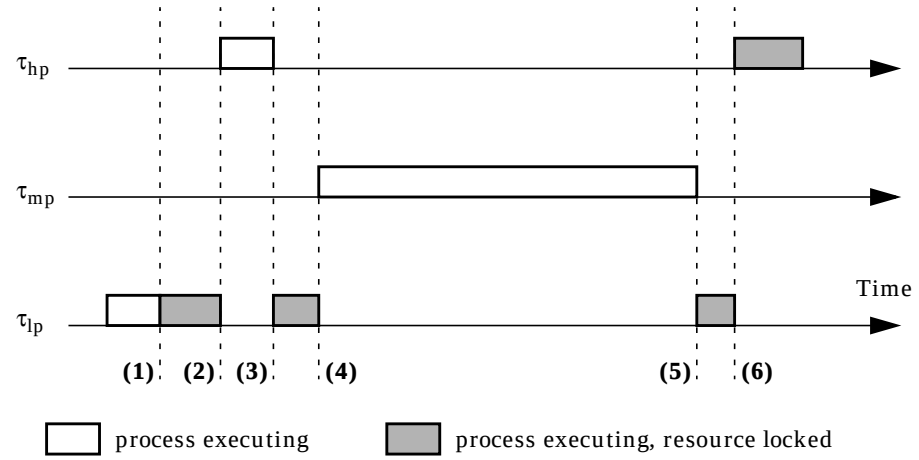
Deadline monotonic scheduling

In most real-time systems, processes exist which have a deadline shorter than the period of the process. This is especially true for systems performing automatic control of technical equipment. As described in Section 2.1.3, such a system samples the state of the controlled equipment at regular intervals. New control signals are computed based on the sample and then output to actuators. It is important that the time from sampling the state to outputting new control signals is short. Otherwise, stability cannot be guaranteed according to the theory of automatic control [ÅW84]. Deadlines shorter than the process periods will thus be common in such systems.

If we change the process model used in RMS to include deadlines shorter than the periods of the processes, we find that RMS is no longer optimal. Instead, the optimal choice turns out to be to assign priorities monotonically with the *deadlines* of the processes. This strategy is called *deadline monotonic scheduling* [ABRW91]. The process with the shortest deadline should be assigned the highest priority and so on. Deadline monotonic scheduling can be viewed as a generalization of rate monotonic scheduling. If all deadlines are set to the respective period, the scheduling strategies will be identical.

Priority inheritance protocols

Priority inversion is a phenomenon where a higher priority process is blocked for an arbitrary long time by a lower priority process. Such a situation occurs when a low-priority process holds a resource that the higher-priority process is requesting. The duration of the blocking by the low-priority process is usually short, but if a third, medium priority, process is released, it will preempt the lower priority process and prevent it from releasing the resource. This can lead to arbitrary delays for the higher-priority process. Priority inversion is illustrated in Figure 2.2.



1. The low-priority process, τ_{lp}, locks a resource.
2. The high-priority process, τ_{hp}, is invoked, preempting τ_{lp}.
3. The high-priority process attempts to lock the already locked resource. It blocks since the resource is already locked by τ_{lp}.
4. A medium-priority process, τ_{mp}, becomes ready to run and preempts τ_{lp}. The high-priority process will be blocked for the entire execution of τ_{mp}, even though τ_{hp} and τ_{mp} does not share any common resources.
5. The medium-priority process is suspended and the execution of τ_{lp} is resumed.
6. The resource is finally released. The high-priority process immediately preempts τ_{lp} and locks the resource.

Figure 2.2 Priority inversion. It is illustrated how a process can be blocked by a lower priority process for an arbitrary long time even though the processes does not share any common resources.

To avoid blocking caused by priority inversion, *priority inheritance protocols* are employed. All of these protocols involve temporarily raising the priority of a process that has allocated a resource. The probably most widely used priority inheritance protocol is the *basic inheritance protocol*. Other protocols are the *priority ceiling protocol* and the *immediate inheritance protocol*. The rest of this section will briefly present these protocols.

The *basic inheritance protocol* [SRL90] states that whenever a process blocks because a resource it attempts to lock is already locked by a process with a lower priority, the process currently possessing the lock will inherit the priority of the blocked process. The priority of a process is thus raised if, and only if, it is blocking a higher priority process. When the resource is released, the priority will be set to what it was before the priority was raised. Processes with intermediate priority levels will thus not be able to prevent a lower priority process from exiting a critical

section, and the maximum blocking time of the higher priority process will be bounded. The protocol is easy to implement and does not require an a priori analysis of the process set and its utilization of semaphores. It does not impose any restriction on how resources are allocated or released.

The *priority ceiling protocol* [SRL90] imposes a more restricted resource locking policy than the basic inheritance protocol does. A process must not hold a resource between executions and resource locking/unlocking must be properly nested. That is, resources must be released in the opposite order to how they were locked. An a priori analysis of which processes use which resources must be made. For each resource, a priority ceiling is computed. This is the priority of the highest-priority process that access the resource. Whenever a process attempts to lock a resource, it is checked whether the priority of the process is strictly higher than the ceilings of all previously locked resources in the system. If it is, the process is allowed to lock the resource. If not, the process is blocked. The process causing the lock inherits the priority of the blocked process.

The priority ceiling protocol can be expensive to implement, but has some appealing advantages: First, a process can only be delayed once by a lower priority process. The length of the delay corresponds to the longest critical section in the lower priority processes. Second, it guarantees that no deadlocks can occur. It does so by allowing resource locking only if it can be guaranteed that locking the resource might not cause a future deadlock. The consequence of the scheme is that processes are, on average, blocked more often than if the basic inheritance protocol is used. Blocking often occurs even though two processes do not attempt to lock a resource simultaneously. The amount of blocking in the average case is thus worse.

The *immediate inheritance protocol* has the same requirements as the priority ceiling protocol, i.e. no resources held between invocations, properly nested locking/unlocking, and an a priori analysis of the process set. However, it is much easier to implement while retaining the attractive worst-case performance of the priority ceiling protocol. It guarantees that no deadlock occur as well. Again, we must assign a priority ceiling to each resource. At run-time, when a process attempts to lock a resource we immediately set the priority of the process to the maximum of the current priority of the process and the ceiling of the resource. The immediate inheritance protocol is described by Lampson and Redell [LR80].

2.2.3 Earliest deadline first scheduling

All the scheduling strategies we have studied so far require that we take some a priori scheduling decisions. With static cyclic scheduling we had to construct an explicit scheduling table and with fixed priority scheduling we had to assign priorities to the individual processes. If we add a new process to the system, or if we change it in some other way, we have to redo this work. A more dynamic approach is represented by *earliest deadline first scheduling* (*EDF*). Here, all the scheduling decisions are delayed until runtime. However, when designing software for safety-

critical systems, we still want to make an a priori analysis of the process set to determine whether all deadlines will be met or not. Schedulability analysis is thus still required before running the system, which lessen the advantage of the strategy over fixed-priority schemes.

EDF states that the processor should be assigned the process that is closest to its deadline. If an external event causes a process to become ready, the system checks whether the deadline of the newly released process is shorter than the deadline of the currently executing one. If so, the currently executing process is preempted. EDF is more flexible than fixed-priority schemes since it is possible to dynamically add new processes to the system without doing a global recalculation of priorities. The scheduler automatically does its best to meet all deadlines.

The optimality of EDF has been proven for a system with arbitrary process invocation and deadline times, and arbitrary and unknown (to the scheduler) WCETs for each process [Der74]. An interesting property of earliest deadline first scheduling is that it is possible to achieve a 100% processor utilization ratio without violating any deadlines, which is typically not the case when fixed priority scheduling is used. However, overload is handled very poorly by earliest deadline first scheduling. Experience has shown that performance degrades rapidly in the case of overload. A domino effect usually results as the scheduler continuously gives priority to processes that are close to missing their deadlines.

2.3 Process scheduling in existing real-time kernels

In order to see what scheduling strategies are used in practice, this section briefly surveys some typical real-time kernels and study what type of process scheduling strategy they use.

Hawk: Hawk [HH89] is a small real-time kernel designed specifically for the SANDAC multiprocessor computer for embedded systems. It has been used in systems for airborne guidance and control as well as for land-based navigation systems. A priority-based dispatcher schedules the processes on each microprocessor. Fixed-priority scheduling is thus assumed, but it is possible for the system to change the priority of a process dynamically. It is also possible to install a special scheduler process in order to implement other scheduling strategies.

QNX: QNX [Hil92] is a commercial real-time operating system built around a small microkernel supporting the basic concurrency functionality. Higher level operating system functionality is implemented by a set of cooperating processes surrounding the microkernel. The process scheduling model is based on preemptive fixed-priority scheduling. The kernel implements the basic inheritance protocol in order to avoid problems with priority inversion.

VxWorks: VxWorks [WRS95] is a commercial real-time operating system. It is clearly the most common operating system within the field of robotics. It was also used on Nasa's recent Pathfinder mission to Mars. The process model is based on fixed-priority scheduling with priority inheritance.

Spring: The Spring kernel [SR89] is a research real-time kernel, developed at the University of Massachusetts, which supports distributed real-time systems. The system attempts to dynamically ensure the schedulability of the system, allowing new processes to be added to the system on the fly if they prove to be schedulable. A new process is assigned to one of the processors and inserted into its system task table, which is an explicit representation of the process schedule.

Ada: The Ada programming language [I+83] was originally developed to meet the requirements on a new universal and standardized programming language issued by the United States Department of Defence. The first ANSI/ISO standard, Ada 83, appeared in 1983. The language was intended to provide a portable language for embedded systems and contains support for concurrent computing. The scheduling model prescribed by the standard was preemptive fixed-priority scheduling. In 1995, a new standard arrived, Ada 95, which provides more flexibility in the choice of scheduling strategy. However, most implementations of Ada 95 use preemptive fixed-priority scheduling.

The JAS 39 fighter: The swedish JAS 39 fighter aircraft is a highly computerized modern fighter aircraft developed by SAAB Military Aircraft. The system computer of the fighter runs a proprietary real-time kernel [Fol93]. The process model consists of a fixed number of periodic processes with harmonic periods. In addition, a low-priority background process exists. The kernel uses fixed-priority scheduling with preemption. Rate monotonic scheduling is used to assign priorities to the periodic processes. The background process has the lowest priority.

2.4 Summary

A real-time system is a system in which not only the output of the system determines success or failure. The *time* at which the output is produced also matters. The system must have a response time to external stimuli that is shorter than a given limit. The system is said to have *deadlines*. The real-time requirements vary between different types of applications. A *hard* real-time system contains processes which must always meet their deadlines. *Embedded systems*, systems where the computer is a part of a larger piece of equipment, are mostly *hard* real-time systems. They typically contain parts with soft real-time demands as well. Embedded systems often perform automatic control of external physical processes and are often safety-critical. It is vital that such systems are *predictable* and that we can perform a priori analysis in order to guarantee schedulability.

Various execution models have been developed for implementation of real-time systems. The first systems used static scheduling techniques, but more dynamic scheduling approaches are used more and more. The most commonly used execution model in real-world real-time systems is fixed-priority scheduling with preemption. It is therefore reasonable to develop a real-time memory management strategy with primarily this model in mind. It is, however, important that it can be generalized to other models as well.

Chapter 3

Automatic Memory Management

This chapter surveys and comments on different approaches to automatic memory management. The real-time properties of existing approaches are reviewed and the aim of the work presented in the thesis is described.

3.1 Introduction

In the early days of software development, there was only *static memory management*. Static memory management means that every entity in the program is statically bound at compile time to a certain memory location. Fortran [ANSI78] is an example of an early programming language that used static memory management. This is a very simple scheme, but it has some obvious disadvantages. First, the size of all data structures must be known in advance. Second, it is not possible to build data structures dynamically depending on input. Third, if we drive static memory management to its point, recursive procedure calls cannot be allowed since the procedure activation records are statically allocated. The latter restriction can be lifted if we introduce a stack for activation records. This approach was introduced in Algol [BMN+60].

The need for more dynamic software soon required a more flexible way of handling memory. A heap on which objects could be allocated dynamically was introduced and with that *dynamic memory management*. The size of data structures could now be allowed to be determined during run time, and data structures could be built as the program executed. *Manual dynamic memory management* was used, which means that the application program is responsible for keeping track of which parts of the heap contain live objects and which parts can be reused for new purposes. Examples of languages supporting manual dynamic memory management are C (*malloc*/*free*) [KR78] and Pascal (*new*/*dispose*) [JW85]. The runtime libraries typically include some support to help the program managing the

heap. It includes operations to manage a free-list containing all free memory segments on the heap. Operations are available to allocate a new object on the heap (*malloc*/*new*) and to return the memory occupied by a no longer needed object to the free-list (*free*/*dispose*). Even so, the program will contain complicated code for controlling when to release objects in memory. Programming errors causing dangling pointers or memory leaks are common.

A way of alleviating the application programmer of the burden of manual memory management, while still retaining the advantages of dynamic memory management, is to hand over the responsibility for determining which memory must be preserved and which can be reused to the runtime system. This approach is called *automatic memory management* or *garbage collection*, *GC* for short. Examples of early uses of garbage collection are Lisp [McC60] and Simula [SIS87]. The rest of this chapter discusses issues related to automatic memory management in general, and automatic memory management for real-time systems in particular. An overview of different techniques is given.

3.2 Memory fragmentation

Fragmentation occurs when objects of different sizes are allocated on the same heap and later deallocated. This leaves holes of free memory interspersed with live objects. A subsequent memory allocation request might not be able to reuse the memory previously deallocated since the holes might be too small to hold the new object.

Robson showed [Rob71] that the total amount of heap store needed to guarantee that an allocation request can always be met will be large if variable block sizes are allowed. Even in the simple case that only blocks consisting of one or two words exist, fifty percent more store is needed compared to the maximum amount of words live at the same time. As the maximum block size grows the worst-case storage needs increase. According to Robson, if the maximum block size is 64 word a heap might be required that is seven times larger than the maximum amount of simultaneously live memory. The impact of variable block sizes on the storage needs is also discussed in [Knu73].

One way of decreasing the problem of memory fragmentation is to use fixed block sizes. There are at least two problems with this approach, namely *internal fragmentation* and the need to *split large objects*. Internal fragmentation arises when small objects do not use the entire memory block they have been assigned. This naturally increases the amount of heap space needed. Large objects, on the other hand, which do not fit into a single block must be split up into several smaller segments. Both time and size overhead will result from the extra management of the segments. If only one type of object (with a fixed size) is allocated on the heap, both of the above disadvantages are eliminated. However, few software systems have this property.

The memory fragmentation problem can be solved by *compacting* the heap. Compaction means that the live objects on the heap are regularly moved and put next to each other, resulting in a single, continuous, area of free memory. Moving the objects and updating all pointers involve some overhead. Memory management algorithms that do not employ compaction are usually said to be *non-moving*.

3.3 Basic garbage collection algorithms

We distinguish between three basic approaches to garbage collection: *reference counting*, *mark-sweep*, and *copying algorithms*. Here, we give a brief presentation of these. For a more exhaustive presentation refer to [JL96], [Wil92], or [Coh81].

The task of the ideal garbage collector is to identify and reclaim the memory occupied by objects that *will not* be referenced by the application program. Since it is impossible for the collector to know which objects that will actually be referenced later, it employs a somewhat more conservative approach approximating the ideal one; identify and reclaim memory occupied by objects that *cannot* be referenced again, i.e. objects that are no longer reachable from the program. The criteria used to decide whether an object is live or not is thus whether it is *reachable* or not.

In this chapter, and in the rest of this thesis, we will use the terminology introduced in [Wad76]. The application program and the garbage collector are viewed as two independent processes sharing a common memory. The first process, the application program, is called the *mutator* since it modifies (mutates) the object graph. The second process, the garbage collector, which is responsible for recovering memory discarded by the mutator, is called the *collector*.

3.3.1 Reference counting

The principle behind reference counting is to store a counter in every object indicating the number of references to the object [Col60].[1] The counters of the affected objects must be updated every time the mutator modifies the object graph. When creating a new reference to an object the counter must be incremented, and when removing a reference the counter must be decremented. When the value of the counter becomes zero, i.e. no references to the object exist, the memory occupied by the object can be reclaimed. Reclaimed objects are typically inserted into a free-list in the same manner as for manual memory managers. The heap is thus not compacted.

1.The counter might be physically present in the object or implicit. The original paper on reference counting [Col60] suggests that objects only referenced once, which is the majority of the objects in many systems, omit the counter in order to save memory space. On the other hand, this necessitates additional checks in connection with each pointer assignment.

The following pseudo code describes how a simple reference counting algorithm might be implemented. The function *NEW* is called to allocate a new object and *SET_POINTER* is used to change the value of a pointer. The procedure *DECREASE_COUNT* is only used internally and is thus not part of the interface to the reference counter. It is assumed that every object contains an attribute called *Count*, which keeps track of the number of existing references to the object.

```
FUNCTION NEW(Size);
   VAR Ptr;
   Search free-list for a suitable memory location. Let Ptr point
      to the new object.
   Update free-list.
   Ptr.Count := 0;
   RETURN Ptr;
END

PROCEDURE SET_POINTER(Ptr,Value);
   DECREASE_COUNT(Ptr);
   Ptr := Value;
   IF Value<>NULL THEN
      Ptr.Count := Ptr.Count+1;
   END
END

PROCEDURE DECREASE_COUNT(Ptr);
   Ptr.Count := Ptr.Count-1;
   IF Ptr.Count=0 THEN
      FOREACH pointer P in object referenced by Ptr DO
         DECREASE_COUNT(P);
      END
      Insert ptr into free-list.
   END
END
```

A nice feature of reference counting is that the GC work is performed in an incremental fashion as the object graph is modified. Long pauses are not very common but can occur once in a while since reclaiming an object may cause other reference counts to become zero and so on. An example of such a situation is when the last pointer to a large tree structure is deleted. All the nodes of the tree will then be unreachable from the mutator and the garbage collector will traverse the entire tree, reclaiming the objects one by one.

A major drawback of reference counting is that circular linked structures of garbage objects cannot be detected. There is always a reference to each object in such a structure, which means that they will not be deallocated. It has been observed that circular object structures is a common phenomenon [BS93]. Reference counting alone is thus not sufficient in long-lived programs. It can be used in combination with some other garbage collection approach capable of recognizing circular structures of garbage, or rely on cooperation from the programmer (e.g. [Bob80]) which is a very unsafe approach. Hybrid algorithms are often used, com-

bining reference counting with mark-sweep traversal. Examples of such algorithms can be found in [Chr84] and [Lin92].

The overhead of managing the counters can be quite high, which limits the use of the method. Every pointer assignment causes counters to be incremented and decremented. A test must also be performed to check for a counter reaching zero, in which case the object must be deallocated. It has, however, been shown that the total cost of counter management can in many cases be reduced significantly by avoiding reference counting operations in special cases [DB76,Bad93], for example when it can be deduced that only one reference exists to an object. Since reference counting algorithms typically do not employ memory compaction, they suffer from problems with fragmentation.

3.3.2 Mark-Sweep

The first mark-sweep algorithm was published in [McC60]. Mark-sweep algorithms make use of two phases; the *marking phase* and the *sweeping phase*. The purpose of the marking phase is to locate and mark all objects that are reachable from the mutator. The second phase, the sweeping phase, traverses the heap examining each object and reclaims the memory occupied by unmarked objects. The sweeping phase may include compaction of the heap, in which case the algorithm is sometimes called *mark-compact*. We will concentrate on compacting algorithms in this thesis and will generally refer to compacting algorithms when we talk about mark-sweep algorithms, unless stated otherwise.

The *mark phase* begins with examining the set of *root pointers*, i.e. the set of pointers located outside the heap, through which all accesses to heap objects are made. Included in the root set are global pointer variables and pointers located on the stack. The objects reached through the root pointers are marked. The contents of the objects marked in this way are then examined, or *scanned*. Objects referenced by pointers within the already marked objects are added to the set of marked objects, if not already marked, and their contents are in turn examined. Some kind of stack is typically used to keep track of objects that have been marked but not yet scanned. In this way the entire graph of reachable objects will be traversed and all live objects will be marked. Algorithms that traverse the object graph in this way to find all reachable objects are also called *tracing* algorithms.

The *sweep phase* reclaims the memory used by the unmarked objects. This is done somewhat differently depending on whether compaction is desired or not. Compaction might be sacrificed in order to achieve lower overhead for memory management, but fragmentation might then cause problems. For non-compacting algorithms, a single traversion of the heap is sufficient, inserting all unmarked objects into a free-list as they are found. Compacting algorithms are more complicated. Several passes might be required. First, the collector must decide on the new locations of the live objects. Then, all pointers must be updated to point to the new locations. The objects must finally be moved to their new locations. The LISP 2 gar-

bage collector [Knu73] uses three passes over the heap to perform these tasks, performing one task in each pass. A mark-sweep LISP 2 garbage collector thus needs to traverse the heap four times to perform a complete GC cycle (one mark and three sweep passes). An improvement to the LISP 2 algorithm was presented in [Tho76]. The mark phase of the algorithm presented there temporarily modifies the object graph in such a way that a linked list is created for each object, containing the locations of all pointers referencing the object. As a result, calculating new locations for the objects and modifying the pointers to the object can be done in a single pass, eliminating one of the sweep passes. Garbage collectors based on this algorithm thus only need to traverse the heap three times in total.

Since the mark-sweep algorithms must process dead objects on the heap as well as live ones during the sweep phase, the time required for one garbage collection will be proportional to the size of the heap.

A case study of a mark-sweep algorithm

In order to gain a better understanding of mark-sweep algorithms, we will study one such algorithm in more detail. The algorithm we have chosen is the LISP 2 compacting mark-sweep algorithm.

Each object on the heap contains a header with room for information needed by the collector during the GC phase. This includes space for a mark bit and the address the object will be moved to during compaction. We consider *Marked* and *NewAddress* to be attributes of every heap allocated object and they contain the required information.

New objects are allocated at the lowest available free address in the heap. Allocation proceeds in this manner until the heap is completely exhausted. At this point, the mutator is suspended and the collector initiates a full GC cycle. The allocation operation can thus be described by the following piece of pseudo code:

```
VAR AllocationPointer; (* Address of next free memory cell *)

FUNCTION NEW(Size);
   VAR NewObject;
   IF AllocationPointer+Size > top of the heap THEN
      MARK_SWEEP();
   END
   NewObject := AllocationPointer;
   AllocationPointer := AllocationPointer+Size;
   RETURN NewObject;
END
```

The GC work, represented by the procedure *MARK_SWEEP*, is performed in four passes. First, a mark pass is performed, setting the mark bit of every object reachable from the root pointer set. This is done recursively starting from the root pointers. Then, three linear sweep passes over the heap are performed. For each

marked (live) object, the first sweep pass calculates the address that the objects is to be moved to during the final compaction pass. The address is stored in the object header. The second sweep pass again traverses the marked objects. Now, the pointers within the objects are updated to reflect the values they will have after the heap is compacted. This is done by dereferencing each pointer, fetching the address in the destination object calculated during the first sweep pass, and finally storing this address in the pointer field that is to be updated. The third pass performs the actual compaction, sliding the marked objects down towards low memory addresses, creating a contiguous sequence of live objects. The mark and sweep process is described by the pseudo code below.

```
PROCEDURE MARK_SWEEP();
   MARK();
   SWEEP_CALCULATE_ADDRESSES();
   SWEEP_UPDATE_POINTERS();
   SWEEP_MOVE_OBJECT();
   AllocationPointer := first free memory cell;
END

PROCEDURE MARK();
   FOREACH pointer P in root set DO
      MARK_OBJECT(P);
   END
END

PROCEDURE MARK_OBJECT(Ptr);
   IF NOT Ptr.Marked THEN
      Ptr.Marked := TRUE;
      FOREACH pointer Son in the object referenced by Ptr DO
         MARK_OBJECT(Son);
      END
   END
END

PROCEDURE SWEEP_CALCULATE_ADDRESSES();
   VAR Ptr,NextFree;
   Ptr := start of the heap;
   NextFree := Ptr;
   WHILE Ptr<AllocationPointer DO
      IF Ptr.Marked THEN
         Ptr.NewAddress := NextFree;
         NextFree := NextFree+OBJECTSIZE(Ptr);
      END
      Ptr := Ptr+OBJECTSIZE(Ptr);
   END
END;

PROCEDURE SWEEP_UPDATE_POINTERS();
   VAR Ptr;
   Ptr := start of the heap;
   WHILE Ptr<AllocationPointer DO
      IF Ptr.Marked THEN
         FOREACH pointer P in the object referenced by Ptr DO
            P := P.NewAddress;
```

```
            END
         END
         Ptr := Ptr+OBJECTSIZE(Ptr);
      END
   END

   PROCEDURE SWEEP_MOVE_OBJECTS();
      VAR Ptr;
      Ptr := start of the heap;
      WHILE Ptr<AllocationPointer DO
         IF Ptr.Marked THEN
            Move object referenced by Ptr to Ptr.NewAddress.
            Ptr.NewAddress.Marked := FALSE;
         END
         Ptr := Ptr+OBJECTSIZE(Ptr);
      END
   END
```

3.3.3 Copying algorithms

Copying algorithms traverse the object graph starting from the root pointers in a manner similar to the mark-sweep algorithms. Thus, they belong to the family of tracing algorithms. However, instead of just marking an object as live when found, it is immediately copied to a new segment of storage. When the entire object graph has been traversed, all live objects have been copied from, or *evacuated* from, the old heap. The old heap now only consists of dead objects and can thus be reclaimed. Copying algorithms are by their nature compacting.

The first algorithm of this type was published in [Min63]. Secondary storage was used to hold the evacuated objects. When all live objects had been evacuated to secondary storage, they were copied back to a contiguous area in the heap. Later variants of copying algorithms use the *semispace* strategy [FY69]. This divides the available memory into two equally sized spaces, used one at a time. When the first semispace is filled up, a GC cycle is performed, evacuating the live objects to the second semispace. This second semispace is now used for allocation of new objects until it in turn is filled up. At this point, the garbage collector moves live objects back to the first semispace and so on. Early copying algorithms used recursion to traverse the object graph, but modern variants use the strategy presented by Cheney [Che70], which uses a *scan pointer* that iteratively traverses the evacuated objects.

The time overhead of copying algorithms is proportional to the amount of live objects, making them attractive compared to mark-sweep algorithms when the ratio between live and dead objects is low. On the other hand, the storage overhead is larger since space is required for two semispaces.

Case study of a copying algorithm

In order to illustrate how copying algorithms work, we will study a semispace algorithm using Cheney's method for traversal of the object graph. An *incremental* copying algorithm will play an important role later in the thesis, but the one we will present here is a non-incremental variation.

The heap is divided into two semispaces, called *tospace* and *fromspace*. New objects are allocated in tospace, as illustrated by Figure 3.1. The pointer *B* keeps track of where objects are allocated. When there is no space left in tospace to hold new objects, the garbage collector is invoked. The garbage collector starts by swapping the meaning of fromspace and tospace. The old fromspace will now became the tospace and vice versa. This is called performing a *flip*. The new fromspace will contain a mix of live and dead objects and the new tospace will be empty. The task of the garbage collector is now to find out which objects in fromspace are still reachable from the mutator, and place them consecutively in tospace. Then, the memory used by fromspace can be reclaimed.

Let us follow a simple example to illustrate how the algorithm works. After having changed the meaning of tospace and fromspace, we assume we have the situation showed in Figure 3.2a, with a total of four live objects. The mutator references the objects on the heap through a set of root pointers, external to the heap but pointing to objects within it. The algorithm assumes that the root pointers are known to the garbage collector. In our example we have two root pointers, *root1* and *root2*.

First, the garbage collector *evacuates* the objects referenced by the root pointers. This is also known as *scanning* the root pointers. When evacuating an object, the garbage collector copies the object to the position in tospace pointed to by B, the allocation pointer. It also stores a pointer to the new copy within the original object for later use, this pointer is called a *forwarding pointer*. Finally, the pointer that triggered the evacuation, in this particular case one of the root pointers, is updated to point to the new copy. If the object referenced by the scanned pointer has already been copied to tospace, the collector merely updates the scanned pointer using the pointer previously stored in the old version of the object. After the root pointers

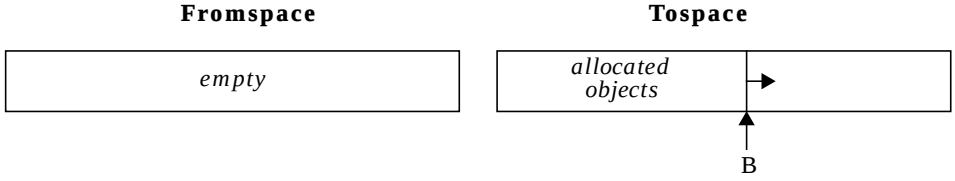


Figure 3.1 The heap structure of a copying garbage collector algorithm during the allocation phase, i.e. while the application program is executing.

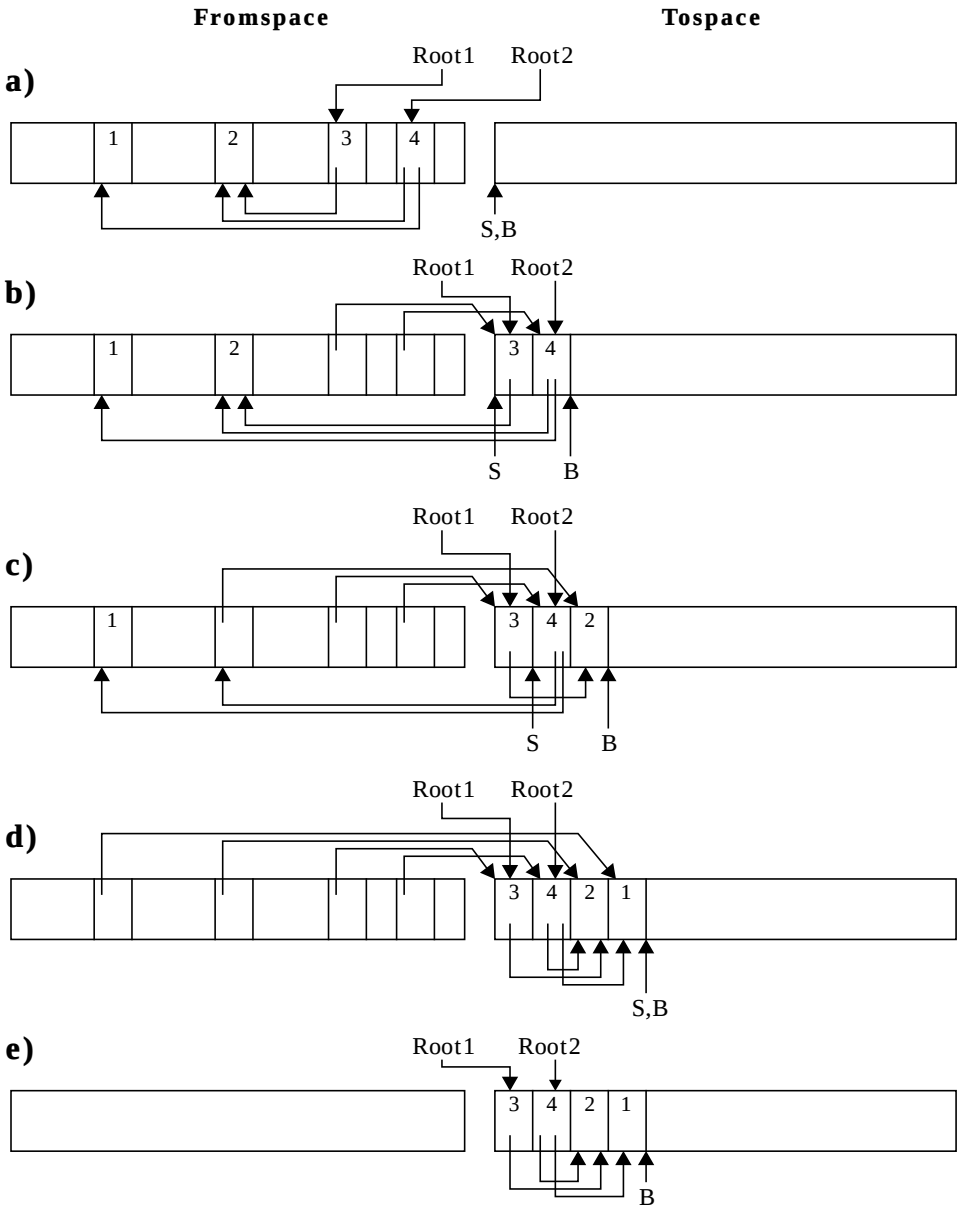


Figure 3.2 A garbage collector cycle for a stop-the-world copying algorithm. Live objects are evacuated to tospace, after which the memory in fromspace can be reclaimed. Allocation can then proceed in tospace until it is again filled up.Then, the meanings of tospace and fromspace are changed and the cycle starts all over again.

have been scanned, the heap is in the state shown in Figure 3.2b. The fromspace versions of evacuated objects are now only used to hold a pointer to the corresponding copy in tospace. Any remaining pointers to the old object will be updated later during the GC process.

After having scanned the root pointers, the garbage collector moves on to scanning the evacuated objects in tospace. The scan pointer, *S*, is used to do this. Starting with the object at the lower end of tospace, the pointers within the objects are scanned one by one as the scan pointer iterates through the objects upwards in tospace. As before, if a pointer to an unevacuated object is found, the referenced object is copied into tospace. The pointer itself is in any case updated to reference the tospace copy. Figure 3.2c illustrates the situation when the pointers of the first evacuated object (referenced by *Root1*) have been scanned. The scan caused another live object to be identified and evacuated. Note that there is still a pointer referencing the fromspace version of the object. This pointer will, however, be updated when the scan pointer, *S*, reaches the object containing the pointer. When scanning the pointer, the collector will discover that the referenced object is already evacuated and will update the pointer using the pointer stored in the fromspace version of the referenced object.

Scanning proceeds until there are no more objects to scan, i.e. until *S=B*. When this happens, the entire graph of reachable objects has been traversed and no live objects remain in fromspace. This is shown in Figure 3.2d. All the memory in fromspace can then be reclaimed (Figure 3.2e) for use during a subsequent invocation of the garbage collector.

A GC cycle is now finished and the next one can begin; the control is handed back to the mutator which resumes its execution. The mutator continues to allocate new objects in tospace. Sooner or later, memory is filled up and the garbage collector is again invoked.

The algorithm can be summarized by the pseudo code below. We assume that heap-allocated objects contain a flag *Evacuated* that indicates whether an object has been evacuated to tospace or not.

```
VAR B,S;

FUNCTION NEW(Size);
   VAR Ptr;
   IF B+Size>top of tospace THEN
      COPYING_GC();
   END
   Ptr := B;
   B := B+Size;
   RETURN Ptr;
END

PROCEDURE FLIP();
   Change tospace to fromspace and vice versa.
   B := start of tospace;
END
```

```
PROCEDURE EVACUATE_AND_UPDATE(Ptr);
   IF Ptr points into fromspace THEN
      IF NOT Ptr.Evacuated THEN
         Copy object referenced by Ptr to address given by B.
         Ptr.Evacuated := TRUE;
         Ptr.ForwardingPointer := B;
         B := B+OBJECTSIZE(Ptr);
      END
      Ptr := Ptr.ForwardingPointer; (* Update the pointer *)
   END
END

PROCEDURE SCAN_OBJECT();
   FOREACH pointer P in object referenced by S DO
      EVACUATE_AND_UPDATE(P);
   END
   S := S+OBJECTSIZE(S);
END

PROCEDURE COPYING_GC();
   FLIP();
   FOREACH pointer P in root set DO (* Evacuate root pointers *)
      EVACUATE_AND_UPDATE(P);
   END
   (* Scan evacuated objects *)
   S := start of tospace;
   WHILE S<B DO
      SCAN_OBJECT();
   END
END
```

3.4 Conservative algorithms

A garbage collector must be able to identify root pointers and pointers within live objects in order to traverse the object graph. Failure to find all pointers may cause live objects to be erroneously considered unreachable and their associated memory will be reclaimed. In most systems, the collector relies on runtime type information provided by the compiler to identify the pointers. In some systems, however, such information is not available to the collector. An example of such an environment is a C or C++ program.

When runtime type information is lacking, some other method must be used to identify pointers. *Conservative* algorithms, of which the one published in [BW88] was one of the first, scan objects, stacks, and global variables with the assumption that every word of memory might contain a pointer. The collector determines whether the potential pointer references a previously allocated object, in which case the object is marked as being live. This strategy means that arbitrary data can be mistaken for pointers, resulting in areas of memory not being reclaimed. Since the collector can never be absolutely sure that a bit-pattern really is a pointer, rather than some other type of data, pointers cannot be updated. Compaction is therefore difficult to achieve in such an environment. One technique to achieve

compaction is to always do pointer dereferencing indirectly through a table of fixed-location forwarding pointers, but this of course assumes a cooperating compiler, or at least that the programmer adheres to strict coding conventions. A compromise is represented by Bartlett's *Mostly Copying Garbage Collector* [Bar88]. This algorithm requires that type information is available for all objects located on the garbage collected heap. Objects referenced only by pointers located on the collected heap can safely be moved and pointers to it updated. If potential pointers to the object are found outside the garbage collected heap, for example on the stack, the object is not moved.

The difficulty of calculating worst-case costs, both with respect to time and storage, for the conservative GC algorithms makes them unsuitable for real-time systems and we will therefore not consider them further in this thesis.

3.5 Generation-based algorithms

Research on the lifetime of heap-allocated objects [LH83,Ben90] has shown that most objects die very shortly after having been created. In contrast, a large portion of the surviving objects tend to live for very long periods. This has inspired people to use different strategies to manage young and old objects in order to avoid moving objects unnecessarily during heap compaction [LH83,Ung84]. The idea is to divide the available storage into two or more smaller areas, or *generations*, which are garbage collected separately. The age of an object controls which generation it is located in. Objects start their lives in the *young* generation. After having survived the initial period of high mortality they are moved, or *promoted*, to an older generation. The process of promoting long-lived objects to older generations is sometimes also called *tenuring*. In this way, garbage collection can to a large degree be isolated to the young generation, which contains a relatively small number of live objects. Moving the bulk of the live objects, the old ones, is thus avoided. A table is maintained for each generation consisting of references to objects in older generations that contain pointers into the current generation. Using this information, a complete traversal of older generations can be avoided each time a GC cycle is performed in a young generation. The old generation (or generations) must also be garbage collected occasionally, but since the allocation (promotion) rate in this part of the heap is quite low this is only necessary with long intervals.

In order to get maximum performance out of a generation-based algorithm, care must be devoted to tuning the parameters of the collector. Factors that must be considered are the number of generations, the generation sizes, and the tenuring policy [UJ88], i.e. when to promote an object to an older generation.

The advantage of generation-based algorithms is a decreased average-case overhead. Most work is isolated to a small young generation which require little time to garbage collect. Most GC-induced delays will therefore be comparably short, in the range of 0.1 seconds or less, and will not be noticed by a human user. The worst-case delays, on the other hand, are typically longer than for single-gen-

eration schemes. Once in a while, tenuring objects will cause a garbage collection in the older generation to follow a collection in the younger generation. It has been proposed that the old generation in turn should be divided into smaller areas that are garbage collected separately. Every once in a while, as objects are tenured, one of the parts of the old generation is scavenged for garbage. In this way, complete traversals of the entire heap are avoided. An algorithm using this approach is the train algorithm [HEM92], which has been implemented in the Beta runtime system [SG95, MMN93].

3.6 Efficiency

The efficiency of a GC scheme can be formulated in several ways depending on the requirements of the application. The amount of storage needed by the garbage collector can be one factor affecting the effiency, as can the time needed. Often a combination of time and space requirements is considered when referring to the efficiency. Since timing is very important in real-time systems, the time overhead for garbage collection is the dominating factor when determining the efficiency for such systems.

Time overhead can be measured in two ways. One way is to look at the total amount of time used for garbage collection. This definition of efficiency is especially appropriate for batch systems where the duration of an individual GC-induced pause is of no interest. The only concern is that the program should (successfully) terminate as quickly as possible. In real-time systems, the total overhead of the scheme is still of interest, but is not a sufficient measure of the efficiency of the garbage collector. Instead, the ability of the collector to comply with the real-time demands of the application must be taken into consideration. This means that the worst-case duration of an individual GC-induced pause is of interest, as is how evenly distributed the pauses are over time.

Algorithms that perform an entire GC cycle in one chunk are often denoted *stop-the-world* algorithms. Stop-the-world algorithms have, in most cases, the lowest total overhead combining low demands on storage with a low GC/mutator time ratio. However, the real-time performance is bad since the individual pauses are unacceptably long.

Comparing mark-sweep algorithms with copying algorithms, one finds that mark-sweep algorithms in general require considerably less memory than copying algorithms. This is due to the fact that the copying algorithms use two equally-sized semispaces, of which only one is actively used at any time. The time overhead for each of the two strategies is somewhat more difficult to determine. If the amount of live objects is large compared to the amount of dead ones, the mark-sweep algorithms are the most efficient. In contrast, if the amount of live objects is small the copying algorithms are more efficient. The time overhead of the mark-sweep algorithms is proportional to the heap size, while the overhead of the copying algorithms is proportional to the amount of live objects.

Generation-based algorithms generally improve the total efficiency by performing garbage collection in only a small part of the heap. Some additional overhead is, however, introduced because of the need to administrate pointers between generations and tenuring information. Since the mortality is high among the objects in the young generation, and thus the number of live objects low, a copying algorithm is suitable for managing these objects. The bulk of objects, those in the old generation, have much lower mortality and can very well be managed by a mark-sweep collector. From a real-time point of view, generation-based algorithms are not acceptable if they introduce long pauses.

3.7 Incremental algorithms

In order to guarantee that a real-time program will meet its deadlines, the system must be predictable. This implies that the garbage collector must have the property that the worst-case delays must be small, bounded, and occur at predictable times. Interactive systems do not have such strict requirements; it is enough that the delays are short enough not to be noticed most of the time.

Incremental algorithms which only perform a very small amount of work during each invocation, such as those presented by Baker [Bak78] and Wadler [Wad76], are often used to achieve automatic memory management for soft real-time systems. The worst-case delay caused by an individual invocation of the collector is typically in the range of 1-10 ms. The algorithms are based on one of the tracing algorithms, dividing the work of a full GC cycle into many small bounded increments, which are executed interleaved with the execution of the mutator. GC work is in most cases triggered by memory allocation and pointer operations.

An incremental real-time version of a mark-sweep algorithm was published by Wadler in [Wad76], but the best known incremental algorithms are based on the copying algorithms. Baker has presented an often used algorithm of this type [Bak78].

A complication caused by the mutator running interleaved with the collector is that it is more difficult to ensure that the collector correctly identifies all reachable objects. The traversal of the object graph performed by the collector is divided into many increments. Therefore, at any time during the execution of the mutator, the set of live objects can be divided into objects not yet found by the collector, objects identified as live but not yet scanned, and objects that have been both identified as live and scanned. The contents of the latter objects will not again be examined by the collector. If the mutator in this situation attempts to store a pointer to an object not yet found by the collector in an already scanned object, the collector might fail to identify the referenced object as being reachable. Baker's algorithm employs a *read barrier* augmenting each operation reading the value of a pointer with a check to ensure that the object referenced by the pointer is marked as live. An improvement of Baker's algorithm which uses a *write barrier* instead of a read barrier (writes are less common than reads) was published by Brooks [Bro84]. Bengtsson

[Ben90] has generalized Brooks' algorithm to handle multiple generations, giving the collector the advantages of the generation-based algorithms while still preserving reasonable real-time properties.

Making copying and mark-sweep algorithms incremental increases the real-time performance of the algorithms, but at the same time introduces new sources of overhead. One such source is the additional work that has to be performed by read and write barriers. Another source of overhead is *floating objects*. It might happen during a GC cycle that an object that has been marked as live by the collector becomes unreachable due to the actions of the mutator. The collector will retain the object and the memory occupied by the object will not be reclaimed during the on-going cycle. The memory will be reclaimed during the next cycle, but floating objects will cause both some storage and time overhead. Bengtsson's thesis [Ben90] contains a thorough analysis of the efficiency of different incremental GC algorithms.

Special properties of the system or programming languages can in some situations be used to achieve performance sufficient for soft real-time systems. An example of this is early versions of the Erlang programming language [ADVW92]. Here, each process was assigned a separate heap. The heap size was in most cases just a few hundred bytes. Due to language properties, there could be no pointers between objects allocated on different heaps; processes communicated by passing copies of data objects to each other. The heaps were garbage collected individually using a stop-the-world policy, but since the heaps were very small so were the individual delays. The technique is, however, not generally applicable to other types of systems: The strategy does not scale up when larger heaps are required. In addition, the overhead for keeping track of references from one heap to another increases quickly as the number of heaps is increased.

3.8 Scheduling properties

Systems that have to comply with real-time demands also put demands on the memory management strategy. The chosen strategy must be efficient enough not to prevent the software from meeting its deadlines. Several levels of real-time demands exist. Some applications may have very relaxed demands while others may have very strict demands. Different memory management solutions might be appropriate depending on the level of real-time demands. This section discusses which memory management strategies are available to meet various timing requirements.

3.8.1 Stop-the-world

The only time-related requirement for a batch system is that the time for the processing should be as short as possible. This implies that a GC strategy that min-

imizes the total amount of time spent on garbage collection should be chosen. There are, however, no restrictions on the size of the individual pauses induced by the garbage collector.

Traditional GC algorithms of the *stop-the-world* type are very suitable for batch systems. Collectors of this type are inactive until the heap is exhausted, i.e. until a memory allocation request from the mutator cannot be satisfied. When this happens, the application program is temporarily halted and a complete GC cycle is performed, reclaiming all dead objects, hence the name stop-the-world.

The individual delays incurred by garbage collection are typically in the range of a few seconds up to several minutes depending on the system. Practically all of the early approaches to garbage collection were based on stop-the-world algorithms, with good reason, since most programs ran in batch mode. Examples of stop-the-world algorithms are [McC60,Tho76,FY69].

3.8.2 Interactive systems

Long GC-induced pauses should not occur in an interactive system, at least not too often, although occasional long pauses might be tolerated. The trick to reduce the duration of the individual pauses induced by the garbage collector is to somehow partition the GC work into smaller chunks. The work can then be spread out over a longer time-span instead of being concentrated to one point in time. One way of partitioning the work is to garbage collect only a limited part of the heap at the time. This is the approach taken by the generation-based algorithms [LH83,Ung84] (described in Section 3.5). In most cases the garbage collection will be isolated to a relatively small part of the heap, namely the young generation, causing pauses no longer than some tenths of a second. Sometimes it will, however, be necessary to garbage collect also the old generation, which will cause considerably longer pauses.

An improvement of the generation-based algorithms, *the train algorithm* [HEM92], addresses the problem of garbage collecting the old generation in a non-disruptive way. Here, the old generation is divided into a number of segments, each garbage collected separately. This scheme reduces the pauses caused by garbage collection in the old generation to acceptable levels [SG95].

Shorter delays can be achieved by dividing the GC work into many small increments. One such incremental strategy is reference counting (Section 3.7), another is the generation-based algorithm presented in [LH83]. Here, small chunks of work are performed in connection with each pointer assignment. Mark-sweep and copying algorithms can also be made incremental, running as coroutines in parallel (actually interleaved) with the mutator, as we will see in the next section. It has also been proposed to let the garbage collector run as a completely parallel process [Wad76,DLM+78,BDS91].

3.8.3 Hard real-time computing

Hard real-time systems have very strict demands on response times. The maximum allowed response times are often 1 ms or less. The consequences of missing even a single deadline are often severe, causing the system as a whole to fail. It must be possible to predict the worst-case behaviour of the software in order to ensure beforehand that all deadlines will be met. Hard real-time systems are thus very sensitive to how GC work is *scheduled*.

Sequential garbage collection

The traditional approach for hard real-time systems is based on fine-grained incremental algorithms. The garbage collector is invoked each time an allocation request is made and performs an increment of GC work. Other pointer-related operations, such as pointer access or assignment may also trigger GC work. It is easy to guarantee that the garbage collector keeps up with the allocation requests since the mutator is suspended until sufficient GC work has been performed.

The incremental real-time algorithms, as described up to now, guarantee small (down to around one millisecond on typical hardware of today) worst-case bounds on the cost of memory management-related operations. However, this is not sufficient for many hard real-time systems, as has been pointed out in for instance [Wit92] and [WJ93]. The problem is that servicing a task will often involve performing a whole series of memory management operations. The cumulative worst-case overhead for garbage collection during the execution of the task may therefore grow quickly, making it impossible to guarantee that the deadline is always met. It is not enough that the duration of a single GC induced delay is small and bounded, but the cumulative overhead for the critical task must also be small. This is difficult to guarantee when traditional incremental algorithms are used. Clustering of relatively long GC-induced delays is an unwanted property of most incremental algorithms, which stresses the problem.

One solution to the problem of rapidly growing cumulative worst-case delays is to try to make the overhead for individual memory management operations practically negligible. One step towards achieving this can be to not compact the heap, thus using non-moving incremental algorithms. In return, the problem of memory fragmentation will have to be addressed in some other way. This approach has been widely used, among others by Baker [Bak92], Wilson [Wj93], and Yuasa [Yua90]. The remaining overhead can still be too high for use in systems with very strict demands on response times, however.

Concurrent garbage collection

GC-induced delays can be made negligible if garbage collection is performed by an execution thread completely separate from the mutator thread. The process responsible for garbage collection is normally given a lower priority than the application processes which are given precedence. Application processes will consequently not be significantly delayed by garbage collection.

The problem with concurrent garbage collection is to guarantee enough progress of the GC work. If the GC process is not assigned a sufficient amount of CPU time, it will not be able to reclaim memory at the same rate as the mutator requests it. The mutator will block until the garbage collector has freed enough memory to satisfy the request, which can result in a violation of a deadline. A global schedulability analysis of the entire system must be performed in order to show that critical processes are never delayed by garbage collection.

Concurrent garbage collection has been proposed by various people, e.g. Steele [Ste75], Dijsktra et al. [DJM+78], and Appel et al. [AEL88]. The problem of scheduling analysis is not given much attention, however. Instead, the work concentrates on algorithmic and synchronization-related issues.

An interesting variant of concurrent garbage collection is presented by Nilsen [NS94]. Here, an incremental copying (and thus compacting) algorithm of the Baker type is utilized. Special hardware is used, which performs the GC work on a separate processor. However, as long as standard mass-produced microprocessors do not include such support, this technique will probably be of limited use.

3.9 Memory hierarchies in real-time systems

The memory system of modern computers is typically implemented as a hierarchy. At the top of the hierarchy we find CPU registers and on-chip memory caches. Further down we in turn find secondary caches and the main memory (DRAM chips). At the bottom we can find mass storage units in the form of disk drives which are used to implement virtual memory. This architecture is motivated by the necessity of making a compromise between hardware cost and performance.

High-performance components such as fast on-chip memory are expensive in comparison to disks or the slower memory chips used for the main memory banks. The bulk of data and code is stored in slow memory, but the system attempts to store frequently used pieces of data and code in faster parts of the memory system in order to minimize the cost of memory access. The result is that the cost of a memory access will vary depending on where in the memory hierarchy the requested piece of information is stored. The difference in cost can be very large, especially if secondary storage (disk drives) is used for demand paging. An access to main memory can easily cost 10 times as much as an access to the processors on-chip cache. The cost will be even higher if data has to be read from disk, perhaps 1000 times more expensive than a cache access. In order to achieve maximum performance, we

want to organize our programs such that as many memory accesses as possible can be serviced by fast memory components. Note that this only minimizes the *average* cost of a memory access. The *worst-case* cost of an access still depends on the slowest part of the memory hierarchy (main memory or secondary storage).

The choice of GC algorithm affects the memory access pattern directly by the accesses performed by the garbage collector. It also affects the access pattern indirectly by controlling where in memory new objects are allocated and (in the presence of compaction) by moving objects around in memory. Poorly constructed GC algorithms can interact badly with caches and virtual memory [Lar77, WLM92, Zor89].

Hard real-time systems

A page fault is generated whenever a memory access cannot be serviced because the requested data has been swapped out to secondary storage. The operating system responds to the page fault by reading in the memory page containing the requested data from disk which may take up to 100 ms, during which time the application program is suspended. Such long delays are unacceptable in practically all embedded systems, which excludes the use of virtual memory. Another reason for not using virtual memory is the desire to avoid using failure-prone mechanical hardware such as disk drives, especially in mobile applications.

Much of the speed of modern microprocessors come from their aggressive use of on-chip caches. However, their use in hard real-time systems can be a mixed blessing [But97]. If the application requires a very high degree of predictability, caching can be a direct disadvantage since some nondeterminism is introduced. Most memory accesses will be cheap, but every now and then a cache miss will require additional time. The worst-case cost for a memory access can very well be higher when a cache is used compared with accessing the main memory directly. This is because the hardware must first check the cache and then access the main memory. Overhead may also be associated with additional house-keeping in order to keep main memory and the cache consistent in connection with memory writes.

GC algorithms which interacts well with caches is desirable for hard real-time systems. However, it is the worst-case behaviour which is of dominating interest. Existing cache-conscious GC algorithms attempt to minimize the total cost of cache misses, i.e. to improve the average case behaviour. Since it is not obvious that the worst-case behaviour will improve, the rest of this thesis will not concentrate on these issues.

3.10 Problem statement

Memory management for hard real-time systems, be it manual or automatic, must take several issues into consideration in order to be feasible:

- *Robustness.* The memory management strategy should aid in producing safe and robust programs. The programmer should be alleviated from writing complex code for managing the memory manually resulting in hard-to-find programming errors.
- *Efficiency.* Processor time is a limited resource, especially in embedded control systems where the CPU usage tends to be high. The memory manager must consequently use the available processor time efficiently and intrude as little as possible on the execution of the application program.
- *Predictability.* Hard real-time systems require predictable behaviour, making it possible to perform an a priori analysis of the worst-case performance. The memory manager must therefore provide strict upper bounds on the associated overhead.
- *Schedulability.* The high demands on reliability and robustness of embedded software often call for a priori schedulability analysis. It must be ensured that memory management does not violate the schedulability requirements.

Automatic memory management is desirable in hard real-time software systems. Introduction of garbage collection reduces the complexity of program code and virtually eliminates hard-to-debug programming errors such as dangling pointers and memory leaks. This results in more robust systems.

Much work has been devoted to finding new efficient algorithms for garbage collection and improving old ones. Even more efficient algorithm can surely be developed, but we do not feel efficiency is the major obstacle for hard real-time garbage collection. Instead, the problem is to achieve enough predictability and to find efficient techniques to make sure that a system will meet all its hard deadlines.

Our goal is thus to find techniques that make automatic memory management, i.e. garbage collection, feasible for hard real-time systems. We primarily concern ourselves with embedded systems for automatic control. We aim at tailoring existing GC algorithms to fit the specific requirements of such systems. Architectural properties of control software can be used to schedule the GC work such that it does not interfere with the control algorithms of the application. Existing techniques for schedulability analysis are adapted to make it possible to perform a priori schedulability analysis.

3.11 Summary

A brief overview of the field of *automatic memory management* was presented. Automatic memory management is characterized by leaving the problem of detecting memory that can be reused for new purposes to the runtime system, or more exactly, to a *garbage collector*. The chapter compares automatic memory management with the more primitive manual memory management which places all responsibility for keeping track of what memory to reuse on the application programmer. The conclusion is that automatic memory management leads to less complex programs with less programming errors, which is of vital importance in embedded systems.

The three basic classes of algorithms for garbage collection were described: reference counting, mark-sweep traversal, and copying algorithms. The latter two types of algorithms were found to be suitable for use in embedded systems in their incremental variants. Incremental garbage collection implies dividing the work of the garbage collector into many small chunks and distributing them evenly over time.

Techniques such as conservative garbage collection, i.e. garbage collection for systems with no or incomplete runtime type information, and generation-based garbage collection were surveyed. Conservative garbage collection was deemed inappropriate for safety-critical embedded systems due to its unpredictable memory usage. Generation-based algorithms promise low average-case overhead, but their worst-case performance is typically worse than that of other algorithms.

Garbage collection has traditionally been difficult to introduce in systems with hard real-time requirements, because of the difficulty to achieve low overhead and enough predictability. A key issue related to the solution of the problem is to develop techniques for how the GC work is *scheduled*. Two scheduling variants for incremental garbage collection were described; sequential garbage collection and concurrent garbage collection. Both variants have their advantages, but also disadvantages that make them less suitable for use in hard real-time systems. There is thus a need for an improved scheduling strategy.

We conclude that an incremental traversal GC algorithm is a suitable choice for use in hard real-time systems. It must, however, be combined with a new scheduling technique in order to meet the demands on predictability and low overhead in critical situations. Furthermore, techniques for a priori scheduling analysis of the collector must exist in order to make it possible to guarantee that a a safety-critical application will always meet its deadlines.

Chapter 4

Scheduling Garbage Collection

In this chapter we describe how to schedule GC work in such a way that high-priority processes are not disturbed. It is shown how an incremental GC algorithm can be modified for our scheduling strategy. We further discuss how to perform scheduling analysis for a system including garbage collection.

4.1 Introduction

Hard real-time systems used for automatic control purposes must satisfy two important timing demands. It must be guaranteed that critical processes can be invoked with a very small latency. Furthermore, critical control processes must finish their execution as soon as possible after invocation in order to minimize the control delay. This implies that the memory management scheme should therefore cause minimal disturbance of such processes.

Good incremental GC techniques exist for *soft* real-time systems. These techniques are, however, not directly applicable to hard real-time systems. In soft real-time algorithms, garbage collection is usually triggered by memory management operations and pointer manipulation. Operations that trigger garbage collection, and thus cause GC-induced pauses, are pointer reads, pointer writes, and memory allocation. The worst-case overhead for each operation might seem to be small enough for real-time requirements applications, but the worst-case overhead of successive operations quickly add up to too long delays, as has been noted in [Wit92] and [WJ93]. A property of many soft real-time algorithms is that long pauses induced by garbage collection are clustered together, making the actual case very close to the worst case. These rapidly accumulating costs for garbage collection make it very difficult to meet tight deadlines.

The high worst-case overhead for pointer manipulation and memory allocation must somehow be eliminated in order to perform garbage collection in hard real-

time systems without significant disturbance. One way to do this is to reduce the amount of work that has to be performed by the garbage collector. Less total work means that less work has to be performed at each invocation of the collector. This in turn means significantly shorter individual delays. The worst-case overhead will therefore not add up as quickly as before. The method used to reduce the required work is often to give up an important property of many GC algorithms, namely memory compaction. The problem of memory fragmentation must then be addressed separately. This approach has been used in many collectors, e.g. [Bak92], [WJ93], and [Yua90].

As we have seen, a garbage collector for hard real-time systems must provide good worst-case performance and guarantee negligible disturbance of high-priority, critical, processes. The scheduling demands of the low-priority processes, on the other hand, are more relaxed and the average-case performance becomes more interesting for these processes. This chapter will concentrate on analysing the worst-case performance, but techniques to achieve good average-case behaviour will also be discussed.

4.2 Semi-concurrent scheduling

The requirement that the work of a hard real-time garbage collector should be scheduled in such a way that it does not disturb critical processes implies that garbage collection should be completely avoided when such processes execute. The absence of GC-induced delays during the execution of critical processes makes it unnecessary to give up memory compaction.

Control systems constitute a large part of the hard real-time systems of today. As we have seen earlier, such systems are built around a few high-priority periodically executing processes with hard real-time demands and a set of low-priority processes with soft real-time demands. The constraints on the design of a garbage collector for such a system are:

- The worst-case cost of individual pointer and memory management operations must be kept very small for the high-priority processes in order to avoid large accumulated worst-case overhead for each invocation.
- The high-priority processes must not be prohibited from starting on time, i.e. a short latency is required. It is thus not feasible to lock the system for extended periods while performing garbage collection.
- The garbage collector should provide good overall efficiency in order to be useful even in heavily loaded systems.

The idea explored in this thesis is to suspend the garbage collector when the high-priority processes are executing. The GC work neglected during the execution of the high-priority processes must then be performed in the pauses between the acti-

vations of the high-priority processes. The remaining time will be split between executing low-priority processes and performing garbage collection motivated by the actions of low-priority processes. We suggest that the low-priority processes use the standard scheduling techniques of soft real-time GC algorithms, i.e. pointer operations and allocation requests trigger garbage collection. Figure 4.1 illustrates how the available CPU time would be used in the presence of one periodically executing high-priority process and one low-priority process.

The strategy can be described by having three levels of priority:

1. High-priority processes.
2. Garbage collection required by the operations of the high-priority processes.
3. Low-priority processes and garbage collection their associated garbage collection work.

In order to avoid starvation among the low-priority processes, the garbage collector suspends its work as soon as it can guarantee that the high-priority processes will not run out of memory. A further natural development would be to assign any idle processor time to garbage collection, adding a fourth level of priority. While this does improve the *average-case* performance of the strategy it does not, in the general case, affect the *worst-case* performance. We will therefore not discuss this optimization in detail.

Garbage collection and process scheduling are in most existing literature seen as separate issues, but in order to implement the described scheduling principle it is necessary to integrate them. The cooperation of the scheduler is needed to trigger garbage collection when the high-priority processes are suspended.

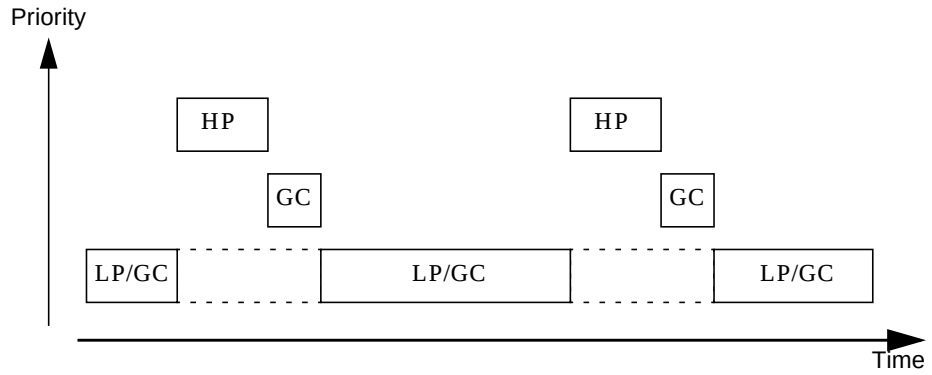


Figure 4.1 Dividing the CPU time between processes. The system consists of one periodic high-priority process (HP) and one low-priority process (LP). Whenever a high-priority process is suspended, the garbage collector (GC) is run. GC work is also interleaved with the low-priority process.

The effect of the proposed scheme is that it will appear to the high-priority processes as if the system was equipped with an ideal memory manager with virtually no overhead. They are never interrupted by garbage collection, nor will garbage collection keep them from being activated at the expected time, provided that enough CPU time remains to run the garbage collector. To the low-priority processes on the other hand, it will appear as if the system had an incremental real-time garbage collector interrupting the application program for short, bounded, periods. GC work will thus be performed concurrently with the high-priority processes and sequentially to the low-priority processes. We therefore call the proposed GC scheduling strategy *semi-concurrent garbage collection*. In the remaining part of this chapter we will investigate how a garbage collector using semi-concurrent scheduling can be implemented and how its behaviour can be analysed.

4.3 Basic garbage collection algorithm

The GC algorithm we will use to illustrate how the collection work should be scheduled is a variant of Brook's algorithm [Bro84], as presented by Bengtsson [Ben90]. Brook's algorithm is in turn a variant of Baker's algorithm [Bak78]. It is an incremental copying algorithm suitable for soft real-time systems, but, as we will see, proper scheduling will also make it usable in hard real-time systems. In this section we will study the original formulation of the algorithm.

4.3.1 Tri-colour marking

We start by introducing the notion of *tri-colour marking*. Originally proposed by Dijkstra et al. [DLM+78], this abstraction is useful when discussing incremental tracing GC algorithms. Heap objects can be in one of three different states as seen by the garbage collector. These states are denoted black, grey, and white. Hence the name tri-colour marking.

- **Black** objects have been identified, and marked, by the garbage collector as being reachable. In addition, the contents of each black object have been scanned for pointers to other reachable objects. The garbage collector has finished examining the black objects and will not visit them again during the present GC cycle.
- **Grey** objects have been identified as reachable, but they have not yet been scanned for pointers to other live objects. A grey object is turned into a black objects when it has been scanned.
- **White** objects have not yet been found by the garbage collector. They may or may not be reachable from the application program. White objects are turned into grey ones as the garbage collector comes across them.

A GC cycle begins with all objects being white. As the garbage collector makes progress, objects are coloured grey and later black. At the end of a GC cycle, all live objects have been coloured black. The unreachable objects were never visited by the garbage collector and are consequently white. The memory occupied by white objects can now be reclaimed.

4.3.2 Algorithm overview

The heap is divided into two equally sized areas denoted *tospace* and *fromspace*, as illustrated by Figure 4.2. New objects are allocated at the top of tospace, at the position denoted by *T*. Allocation proceeds in this way until tospace is filled up. Then, a *flip* is performed, changing the meaning of tospace and fromspace. The old tospace now becomes fromspace and vice versa. Fromspace will contain a mix of live and dead objects. The live objects must be moved, *evacuated*, from fromspace into tospace in order to enable a future flip. The evacuated objects are placed at the bottom of tospace, at the location denoted *B*. The evacuation procedure is performed incrementally as new objects are allocated at the top of tospace. When no free memory remains in tospace, another flip is performed, effectively reclaiming the memory occupied by dead objects. Another GC cycle is now initiated, evacuating the live objects from the new fromspace. Enough evacuation work must be performed in connection with each allocation request to guarantee that all live objects in fromspace have been evacuated before tospace is filled up. Otherwise, the system will find itself in a "catch 22" situation: A flip cannot yet be performed since some objects remain to be evacuated from fromspace, but a flip must be performed in order to free the memory necessary to evacuate the objects.

Since garbage collection is performed in short increments interleaved with the execution of the application program, each increment must leave the heap in a consistent state. When an object is moved, all the pointers referencing the object must be updated to point to the new copy. Since finding all these pointers and updating them all at the time of evacuation can be very expensive, an indirection scheme is used in order to allow the application program to access an evacuated object through both updated and not yet updated pointers. The header of each object contains a *forwarding pointer* pointing to the newest version of the object. When an object is evacuated, the forwarding pointer in the old, fromspace, copy of the object

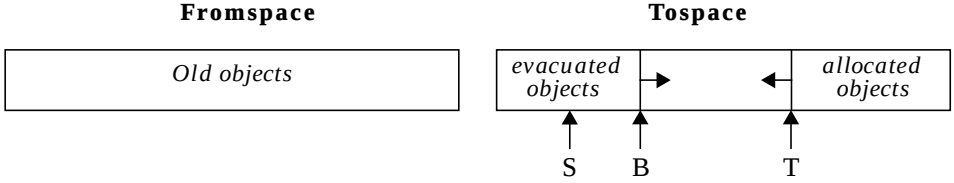


Figure 4.2 The heap structure of Brook's algorithm.

is set to point to the new, tospace, copy. The forwarding pointer of the tospace copy is set to point to itself. All pointer dereferencing is done via the forwarding pointer of the objects pointed to. Dereferencing updated and not yet updated pointers will in this way yield the same result, namely the tospace copy of the object, as illustrated in Figure 4.3. All pointers are updated eventually.

4.3.3 The collector

The garbage collector and the application program, the mutator, are viewed as coroutines. The collector consists of an endless loop performing one GC cycle in each iteration of the loop. A cycle consists of two phases. First, the pointer graph is traversed and all objects found are evacuated. Then, the collector waits until tospace is filled up, after which a flip is performed.

The pointer traversal starts by examining the *root pointers*. The objects referenced by root pointers are evacuated into tospace. Evacuating an object turns it grey according to the tri-colour marking terminology. Not yet evacuated objects, i.e. objects located in fromspace, are considered to be white. The evacuated objects, the grey objects, are scanned next. The pointers within these objects are traced and the referenced objects are evacuated. The grey objects are turned into black objects as their contents are examined. When no grey objects remain, all live objects have been copied to tospace. The recursive scanning of evacuated objects is implemented using a *scan pointer*, denoted *S* in Figure 4.2, sweeping across the evacuated objects starting at the bottom of tospace. The objects referenced by the scan pointer are searched for pointers. When a pointer to a white object is found, the object is copied to tospace and the examined pointer is updated to point to the new location. The scan pointer effectively marks the boundary between black and grey objects.

The collector is described by the pseudo code fragment below. Control is transferred to the application program, and the collector is suspended, whenever

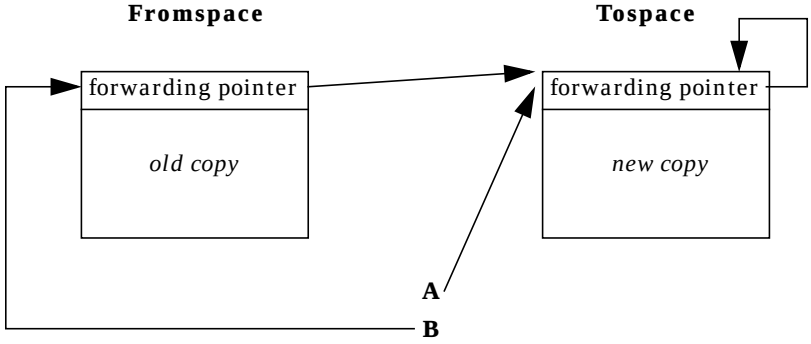


Figure 4.3 The indirection scheme of Brooks algorithm. Dereferencing the updated A pointer or the not yet updated B pointer yields the same result, the tospace copy of the object, after following the forwarding pointer.

enough GC work has been performed and the heap is in a consistent state. Note that the pseudo code below does not explicitly implement these semantics. Execution of the collector coroutine is resumed in connection with allocation requests made by the application program.

```
VAR B;   (* Reallocation pointer for evacuated objects *)
    S;   (* Scan pointer *)

(* Coroutine body *)
LOOP
   WHILE tospace is not full DO
      suspend collector;
   END
   Flip semispaces;
   B := Address of lower end of tospace;
   ScanRootPointers;
   S := B;   (* Address of first evacuated object *)
   WHILE S<B DO
      ScanObject(S);
      S := Address of next evacuated object;
   END
END

PROCEDURE ScanRootPointers;
   FOREACH root pointer DO
      IF root pointer points into fromspace THEN
         IF object referenced by root pointer is unevacuated THEN
            EvacuateObject(root pointer);
         END
         Update root pointer to point to the tospace copy;
      END
      IF enough work performed THEN
         suspend collector;
      END
   END
END

PROCEDURE ScanObject(Object);
   FOREACH pointer in Object DO
      IF pointer points into fromspace THEN
         IF object referenced by pointer is unevacuated THEN
            EvacuateObject(pointer);
         END
         Update pointer to point to the tospace copy;
      END
      IF enough work performed THEN
         suspend collector;
      END
   END
END
```

```
PROCEDURE EvacuateObject(Object);
   Copy Object to the location pointed to by B;
   B := B + size of Object;
   Set forwarding pointer in fromspace copy to point to the
      tospace copy;
   Set forwarding pointer in tospace copy to point to
      the tospace copy itself;
END
```

4.3.4 The mutator

Pointers to evacuated objects, black or grey, can point either directly to the tospace copy of the object or to the old, fromspace, copy. All dereferences of a pointer are therefore made via the forwarding pointer in the objects. This constitutes a very simple *read barrier*.

Since the mutator executes interleaved with the collector, it must make sure it does not introduce pointers to fromspace objects into black objects, since these will not again be visited by the collector. If this was allowed, the collector could fail to identify fromspace objects as being alive. For example, assume that a single pointer exists to a white object somewhere in memory. If the mutator writes a copy of the pointer into a black object and then erases the original pointer, the white object would never be reached by the collector and consequently erroneously considered to be garbage. The mutator must therefore enforce the following invariant:

Invariant: *Black objects do not contain direct pointers to white objects.*

A *write barrier* is used to guarantee that the invariant always holds. Assignments to pointers are monitored by the write barrier and attempts to violate the consistency of the GC scheme are caught. If the new pointer value references a white object the object is immediately evacuated, turning it into a grey object. Assignments to root pointers are monitored in the same way as assignments to pointers located on the heap. The write barrier can be described by the following piece of pseudo code:

```
PROCEDURE PointerAssignment(Pointer,NewValue);
   IF NewValue points into fromspace THEN
      IF object referenced by NewValue is unevacuated THEN
         EvacuateObject(NewValue);
      END
      Update NewValue to point to the tospace copy;
   END
   Pointer := NewValue;
END
```

Allocation requests trigger an increment of GC work by transferring control to the collector coroutine. In the original formulation of the algorithm, the amount of GC work performed is proportional to the amount of requested memory. The work

is performed in immediate conjunction with the allocation request. Before the new object is returned to the requester, the contents of the object are initialized in order to ensure that all pointer fields have consistent values.

```
FUNCTION Allocate(ObjectSize);
   Calculate required GC work;
   Resume collector coroutine;
   T := T - ObjectSize;
   Initialize the contents of the new object;
   RETURN T;
END
```

4.4 Scheduling the garbage collection work

This section deals with how the work of the incremental copying GC algorithm described in Section 4.3 is scheduled and how the collector is synchronized with the mutator. The necessary modifications of the original GC algorithm are described.

4.4.1 Object initialization

The pointer fields of a newly allocated object must have well-defined initial values, because the garbage collector might otherwise misinterpret random bit-patterns as valid pointers when scanning the object. We achieve this by initializing all of the memory cells of new objects to zero. This strategy is simpler to implement than only initializing the pointer fields and has also the advantage that all non-pointer fields are given well-defined initial values as well, which reduces the risk for programming errors.

An obvious strategy for initializing the contents of new objects is to perform the initialization in connection with allocation. When a new object is allocated, it is also initialized before it is passed on to the mutator. The cost of initialization would thus burden the mutator, which is not desirable for the high-priority processes of a control application. Our goal is to minimize the cost of memory management for high-priority processes. As shown in Chapter 6, the cost of memory initialization can easily stand for the majority of the total cost for a memory allocation request.

Our solution to the object initialization problem is to move the responsibility for initializing memory allocated by high-priority processes to the garbage collector. The garbage collector must always ensure that enough free memory is initialized and available for allocation to meet the requirements of the high-priority processes. Low-priority processes, on the other hand, trigger initialization work in connection with allocation requests. The initialization strategy is analogous to the strategy we use for scheduling the rest of the memory management work.

The proposed memory initialization strategy is illustrated in Figure 4.4. The amount of memory that the garbage collector must keep initialized in order to meet the allocation needs of the high-priority is denoted M_{HP} and is derived from the

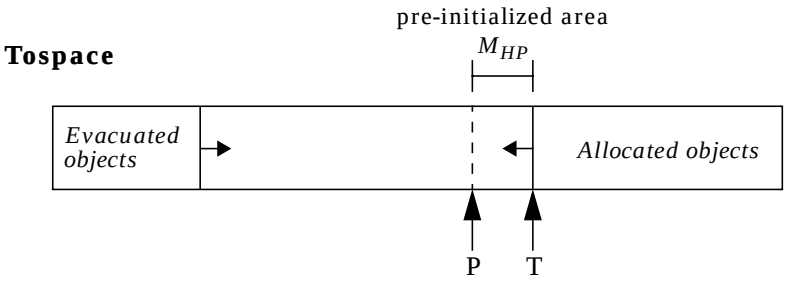


Figure 4.4 The contents of an area of tospace, between the initialization pointer *P* and the allocation pointer *T*, is kept initialized to zero by the garbage collector. This implies that an allocation request made by a high-priority process can always be met without having to spend time on initializing the contents of the new object. The pre-initialized area must be large enough to hold all objects allocated by high-priority process before the garbage collector gets an opportunity to initialize new memory (moving P to the left).

worst-case allocation needs of the high-priority processes. We describe how to determine the value of M_{HP} in Section 4.8.3. New objects are allocated at the position in tospace referenced by the allocation pointer *T*. The initialization pointer *P* refers to the lowest memory cell that has currently been initialized.

Allocation in high-priority processes

The cost for allocation in high-priority processes is bounded and very low since the contents of the new object do not have to be initialized and no garbage collection work is performed in connection with the allocation. An allocation involves only moving the allocation pointer T and writing garbage collection information (e.g. forwarding pointer and object size) into the object header, as illustrated by the following piece of pseudo code.

```
FUNCTION HP_Allocate(ObjectSize);
   T := T - ObjectSize;
   Initialize object header information;
   RETURN T;
END
```

A separate process, *the high-priority garbage collection process* (described in Section 4.4.3), will be invoked whenever a high-priority process terminates and no other high-priority process is ready to run. This process will perform the initialization work and garbage collection work motivated by the allocations performed while the high-priority processes were running.

Allocation in low-priority processes

When a low-priority process allocates memory, an amount of memory equal to the size of the requested object is initialized before the allocation is performed. This is done in order to guarantee that enough pre-initialized memory is always available for allocation in high-priority processes. The pseudo code below describes allocation in low-priority processes.

```
FUNCTION LP_Allocate(ObjectSize);
   WHILE GC work is required OR (P>B AND P>T-M_HP-ObjectSize) DO
      WHILE P>B AND P>T-M_HP-ObjectSize DO
        P := P-1;
        MEMORY(P) := 0;
      END;
      IF GC work is required THEN
         Perform an increment of GC work;
      END
   END
   T := T - ObjectSize;
   Initialize object header information;
   RETURN T;
END
```

The end of a GC cycle - performing a flip

The proposed strategy for memory initialization implies that an area of size M_{HP} in tospace must be initialized before allocation proceeds after a flip. The initialization of this area cannot be performed in connection with the flip since it would bring with it a too long atomic delay. Our solution to this problem is to let the garbage collector (incrementally) initialize M_{HP} bytes of memory in fromspace as soon as it has finished evacuating the live objects from fromspace (at the *end* of the GC cycle instead of at the beginning). The required initializing work is thus added to the total amount of work that has to be performed by the garbage collector during one GC cycle. When the flip is performed, fromspace becomes tospace and the required amount of initialized memory will now be located in tospace.

4.4.2 Lazy evacuation

To avoid the large worst-case overhead for high-priority processes, we employ a *lazy evacuation* scheme. The idea is to delay the actual evacuation of an object until such a time when no high-priority process is executing. At the time of the pointer assignment we only reserve space for the object in tospace, update some housekeeping information, and set the pointer to refer to the reserved area.

Our lazy-evacuation scheme is similar to the one used by Nilsen in his hardware-assisted garbage collector [NS94]. Lazy evacuation has previously been proposed by Baker among others [Bak78]. The purpose of the scheme is to elimi-

nate long unpredictable delays caused by object copying in connection with pointer assignments. The copying is delayed until normal GC work is motivated, i.e. after high-priority processes have finished running or low-priority processes request more memory.

When the write barrier detects that a pointer assignment would introduce a new pointer into fromspace, it checks whether the referenced object has already been evacuated. If so, the pointer is merely updated using the forwarding pointer of the fromspace copy to point to the tospace copy. On the other hand, if the object has not been evacuated, the write barrier checks whether space has been reserved in tospace for the object or not. A flag word used by the collector in the head of the object doubles as a pointer to such a reserved area. If no space has been reserved yet (Figure 4.5a), an area in tospace is reserved for the object. The forwarding pointer of the tospace copy is set to point to the original, fromspace, copy of the object. We thus introduce *forwarding pointers pointing into fromspace*, something that does not occur in the original formulation of the algorithm. Dereferencing a pointer to the tospace copy will now, after following the forwarding pointer, access the fromspace copy. The flag word of the fromspace copy is finally set to point to the reserved area. The final result will be as shown in Figure 4.5b.

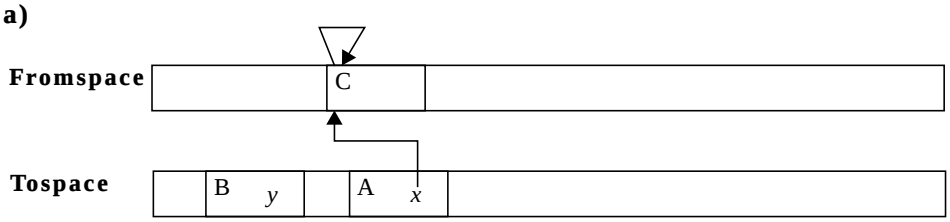
If memory has previously been reserved for the object in tospace, the pointer being assigned to is set to point to the tospace area using the reserved-area pointer in the object.

The actual copying of the object will be performed as a part of the GC work motivated by allocation requests. After copying the object, the forwarding pointers of both copies of the object are set to point to the tospace copy, which will now be the valid one. This is illustrated in Figure 4.5c.

The write barrier can be described by the following piece of pseudo-code:

```
PROCEDURE PointerAssignment(Pointer,NewValue);
   IF NewValue is pointing into fromspace THEN
      IF object referenced by NewValue is unevacuated THEN
         IF NewValue object is not scheduled for evacuation THEN
            Allocate Space for the object in tospace;
            Set the forwarding pointer in the tospace area to
               point to the fromspace copy;
            Store pointer to the tospace area in fromspace object;
            Set NewValue to point to tospace area;
         ELSE
            Set NewValue to point to the tospace area using
               previously stored pointer;
         END
      ELSE
         Update NewValue to point to the tospace copy using
            forwarding pointer;
      END
   END
   Pointer := NewValue;
END
```

a)

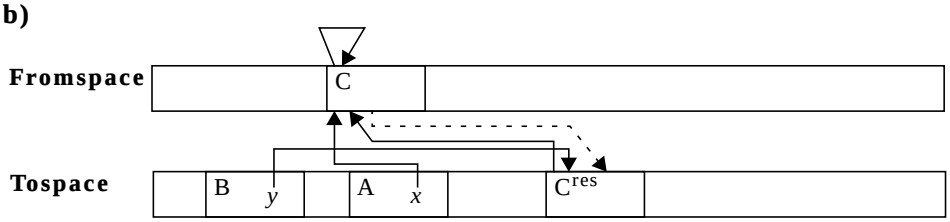


Before a high-priority process attempts to perform the assignment

```
B.y := A.x
```

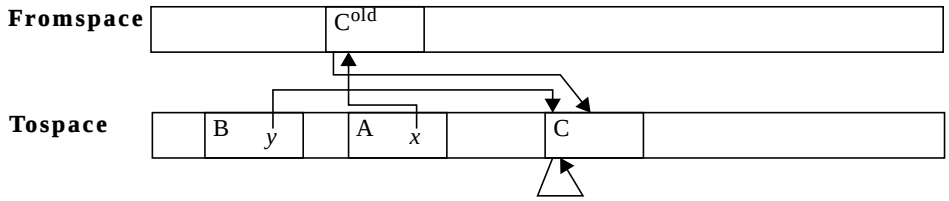
The write barrier catches the assignment since the C object is not previously evacuated or scheduled for evacuation.

b)



After having reserved a new area, C^{res}, for the C object in tospace. A temporary pointer (dotted) to the reserved area makes it possible to avoid reserving multiple areas for the same object. The contents of C^{res} have not yet been initialized, but the forwarding pointer scheme guarantees that it will not be accessed.

c)



When the high-priority process is suspended, the garbage collector finishes the evacuation of the C object copying it to the reserved area C^{res} and setting the forwarding pointers to point to the new location. A.x will be updated to point to the new copy later when the A object is scanned by the garbage collector.

Figure 4.5 The lazy evacuation scheme. It is shown what happens when the write barrier of a high-priority process catches a pointer assignment that could introduce a pointer into fromspace in the scanned part of tospace.

Eventually, the garbage collector starts performing GC work. Whenever invoked, either by the high-priority GC process or by a low-priority process allocating memory, the garbage collector checks whether any objects exist that have been scheduled for evacuation by the lazy evacuation scheme. If so, the evacuation of these objects is commenced.

A pointer into tospace, denoted $B^{unevacuated}$, is used to keep track of the beginning of the area in tospace reserved for evacuation of fromspace objects. The write barrier increments B (Figure 4.2) each time another object is scheduled for evacuation. When the garbage collector is invoked, we have the situation in Figure 4.6. The collector starts to traverse the area between $B^{unevacuated}$ and B, copying the objects pointed to by forwarding pointers in the reserved area into tospace. As objects are evacuated, $B^{unevacuated}$ is updated to reflect the new situation. The transition from Figure 4.5b to Figure 4.5c illustrates this step.

We have chosen to use the same approach for both high-priority and low-priority processes for reasons of simplicity. The low worst-case cost for a pointer assignment achieved by the lazy-evacuation scheme is really only necessary for the high-priority processes. However, using the same approach for all types of processes reduces the amount of machine instructions that have to be inlined at every pointer assignment site. A priority test is eliminated and only one code version must be generated.

4.4.3 The high-priority garbage collection process

As have been noted earlier, no GC work is performed while high-priority processes execute. The work is instead delayed until no high-priority processes are eligible for execution. A special process, the *high-priority garbage collection process*, is responsible for performing the GC work that was omitted when the high-priority processes were executing. The process is also responsible for initializing free memory to zero, such that high-priority processes can allocate memory without having to initialize it at the time of allocation. The process has a priority that is lower than

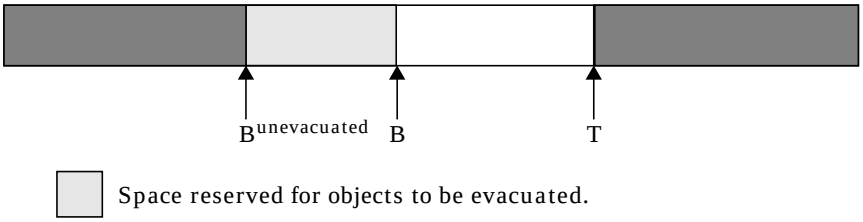


Figure 4.6 Tospace at a point when space has been reserved for evacuation of a number of objects, but at which the objects have not yet been evacuated.

any of the high-priority processes but higher than any of the low-priority processes. It can be described by the following piece of pseudo code:

```
PROCESS HighPriorityGarbageCollection();
BEGIN
   LOOP
      Suspend this process.
      WHILE GC work is required OR (P>B AND P>T-M_HP) DO
         WHILE P>B AND P>T-M_HP DO
           P := P-1;
           MEMORY(P) := 0;
         END;
         IF GC work is required THEN
            IF B^unevacuated<B THEN   (* i.e. evacuation pending? *)
               Evacuate an object.
               B^unevacuated := B^unevacuated+size of evacuated object;
            ELSE
               Resume GC coroutine.  (* See Section 4.3.3.*)
            END
         END
      END
   END
END
```

In order to activate the high-priority GC process after the execution of high-priority processes, some support from the process scheduler is required. Whenever a high-priority process is suspended and no other high-priority process is in a runnable state, the scheduler checks whether any GC work is pending. If so, the high-priority GC process is invoked. When the high-priority GC process is done, low-priority processes are allowed to execute.

4.4.4 Distribution of GC work

Up to now we have only studied how and when GC work is triggered. Virtually nothing has been said about how much work to perform in each increment This section expands on this issue.

To decide whether the garbage collector should work or suspend itself, we a priori calculate a *minimum GC ratio*. The idea is that if the amount of performed GC work is always above this ratio, it is guaranteed that fromspace is completely evacuated before tospace fills up. We denote by W_{max} the amount of GC work necessary in the worst case to evacuate all live objects from fromspace and to initialize M_{HP} bytes of memory (see Section 4.4.1), thus finishing a GC cycle. The minimum amount of memory available in tospace for allocation of new objects immediately after a flip is denoted F_{min}, as illustrated by Figure 4.7.

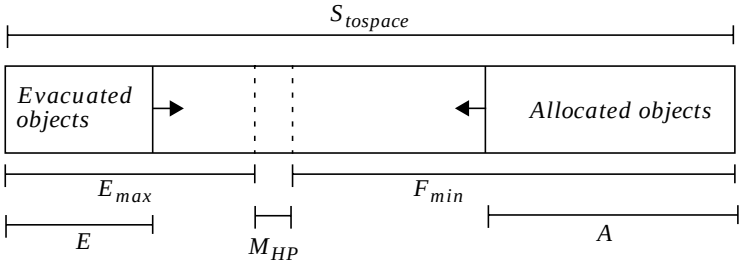


Figure 4.7 The structure of tospace. A minimal area, M_{HP}, is kept available for the high-priority process at all times.

The minimum GC ratio, or GCR_{min}, is defined as:

$$GCR_{min} = \frac{W_{max}}{F_{min}} \tag{4.1}$$

We furthermore define the *current GC ratio*, denoted *GCR*, as the ratio between performed GC work, *W*, and the amount of new, allocated, objects in tospace, *A*:

$$GCR = \frac{W}{A} \tag{4.2}$$

Allocation will cause *A* to increase, while GC work will increase *W*. During the GC cycle (until all live objects have been evacuated from fromspace) the garbage collector performs enough work to make sure that the current GC ratio is higher than, or equal to, the minimum GC ratio. That is:

$$GCR \geq GCR_{min} \tag{4.3}$$

In this way we guarantee that fromspace will be empty before tospace is filled, even in the worst-case scenario. This strategy is essentially the same as in [Bak78] and used in numerous garbage collectors.

Allocation of memory by *low-priority processes* is checked to guarantee that the present GC ratio does not drop too low, i.e. below GCR_{min}. If it threatens to do so, the garbage collector is given priority. The actual allocation of the new object is not performed until sufficient GC work has been performed. The upper bound on the GC work performed in connection with an allocation will be proportional to the size of the allocated object.

If a *high-priority process* is activated shortly before a semi-space flip is due, the remaining memory in tospace could potentially be too small to hold both the objects allocated by the high-priority process and the last objects to evacuate from fromspace. We therefore reserve an amount of memory in tospace, derived from the allocation needs of the high-priority processes, denoted M_{HP}. M_{HP} must be large enough to hold all new objects allocated by the high-priority processes while the

garbage collector finishes the current GC cycle. We will study how to calculate M_{HP} in a later section. Denoting the size of tospace $S_{tospace}$, and the maximum amount of simultaneously live memory E_{max}, the minimum amount of memory available for allocation of new objects, F_{min}, is thus calculated as:

$$F_{min} = S_{tospace} - E_{max} - M_{HP} \tag{4.4}$$

In this way, we will have a buffer area for use by the high-priority processes if necessary. We can furthermore guarantee that the evacuation of fromspace will be finished before tospace is filled up so a semi-space flip can be performed. The high-priority processes allocate memory in tospace *before* the corresponding amount of GC work has actually been performed. The total amount of allocated objects cannot be more than F_{min}+M_{HP} immediately before a flip. The buffer thus guarantees that the amount of new objects, A, will always be smaller than $S_{tospace}$-E_{max}, i.e., there will always be enough memory in tospace to complete the evacuation of fromspace. The cost of this scheme is that the GC rate will be somewhat higher than it would have been if all GC work was performed in connection with allocation operations, see Figure 4.8.

Summarizing Equations (4.1) to (4.4), we get the following expression for how much GC work that must have been performed as a function of the amount of allocated memory:

$$W \geq \frac{W_{max}}{S_{tospace} - E_{max} - M_{HP}} \cdot A \tag{4.5}$$

Since we are using worst-case estimates to calculate how much work to perform, the evacuation will normally be finished well before tospace fills up. We therefore delay the flip until the amount of available free memory drops too low. After execution of the high-priority process, a flip is performed if the remaining free memory is too small to guarantee that the high-priority processes can continue to execute without running out of free memory, i.e. smaller than M_{HP}:

$$S_{tospace} - E - A < M_{HP} \tag{4.6}$$

If servicing a memory allocation request made by a low-priority process would cause the remaining free memory to be smaller than M_{HP}, a flip is triggered before servicing the request.

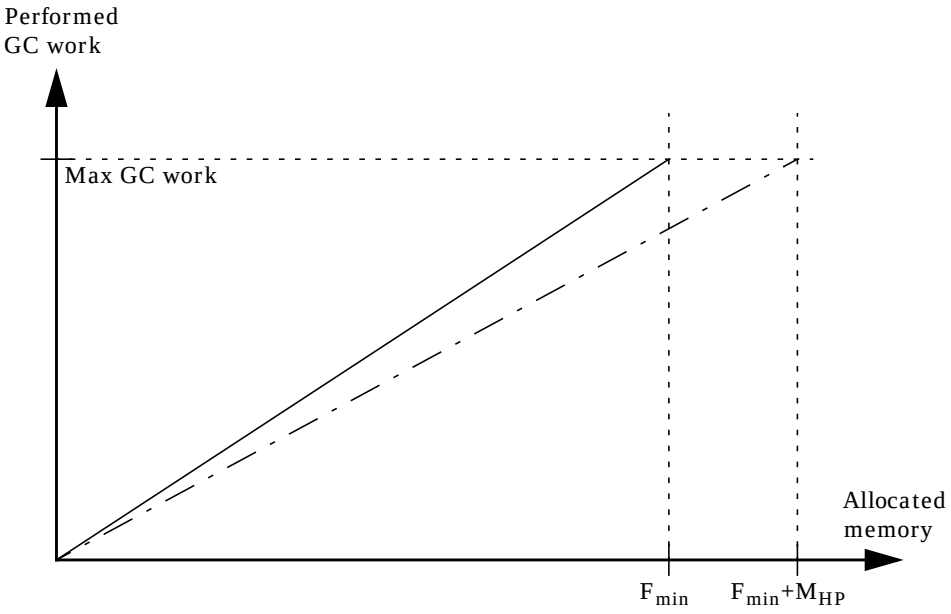


Figure 4.8 Required performed GC work as a function of the amount of newly allocated objects in tospace. As high-priority processes may cause the current amount of performed work to be temporarily below the required amount, we must compensate for this by performing GC work in a slightly higher rate in order to avoid deadlock.

4.4.5 Synchronization

Garbage collection is performed in parallel with the execution of the application processes. In order to obtain proper interaction, the work of the garbage collector must be synchronized with the application processes.

Atomic operations

The systems we study in this thesis are based on a preemptive process scheduler. This implies that the execution of a process may be interrupted at any time, and control is then transferred to another process. It is possible that the interrupted

process was in progress of dereferencing a pointer, or fetching or storing a value from a heap object. If the actions taken by other processes cause the garbage collector to move the object while the previously preempted process is still suspended, resuming the execution will cause it to access a now invalid location in the heap. We must therefore protect pointer operations and heap accesses from being interrupted. Since operations of this kind can be kept very short we suggest that preemption is prevented by disabling the hardware interrupts during these operations.

Interruptible garbage collection

One of the demands on the design of the garbage collector is that it should not prevent a high-priority process from being activated on time. When time comes to activate such a process, garbage collection might be in progress. The cause of this could be that a low-priority process has made an allocation request or the high-priority GC process might be cleaning up after previous allocation requests by high-priority processes. In order to avoid long latencies caused by waiting for the garbage collector to finish, we want such GC work to be interruptible. Garbage collection should not significantly contribute to the total latency of a high-priority process. The maximum delay should be in the order of 50 μs or less for demanding control applications.

The granularity by which the garbage collector can be interrupted can be adapted to the requirements of the application. For many applications, it is sufficient that the collector is allowed to continue executing until the next time the heap is in a consistent state. The most time-consuming operation which causes the heap to be temporarily inconsistent is copying an object from fromspace to tospace. The worst-case latency of a high-priority process would in that case be proportional to the size of the largest object on the heap. If shorter latencies are required, we abort on-going copying. The copying of the object is restarted from the beginning of the object the next time the collector is invoked, since a high-priority process might have modified the contents of the already copied part of the fromspace object. The total worst-case overhead for the garbage collector will then be slightly higher since a double evacuation might be required in connection with each activation of a high-priority process. The worst-case latency will, however, be shorter and it will be independent of the maximum object size.

An alternative approach would be to limit the size of a single object and to implement large objects as several smaller objects. This would, however, slightly increase the time to access large objects due to an extra indirection step. Baker [Bak78] proposed that a read barrier should catch accesses to partially copied objects and access either the old or new version of the object depending on how much of the object had been copied. This requires considerable extra overhead for pointer accesses if implemented in software [NS94]. A third alternative is to access the old object, but maintain a mutation log for all writes to the already copied part

of the object, as is suggested Nettles and O'Toole [NO93] among others. The mutation log is used by the collector to ensure that all writes to the old version of the object is replicated to the new version before the new version is made the current version. The strategy adds to the overhead of the write barrier and also introduces an element of unpredictability since it is difficult to provide an upper bound on the time it takes to go through the mutation log. New writes can be logged by the mutator as the collector works it way through the log.

We always access the old version of an object until it has been completely copied and deal with the threat of inconsistency by restarting the object copying. This yields little mutator overhead for reads and writes. Progress is guaranteed with the help of schedulability analysis. We know that object copying can be aborted at most once every time a high-priority process is invoked, which means that the overhead for restarted copying is bounded. The worst-case overhead is proportional to the maximum object size. Very large objects, e.g. large bitmaps, can in combination with a high CPU load yield a too high overhead, making it impossible to guarantee schedulability. In such a case, it will be necessary to use an alternate strategy, perhaps splitting large objects or storing them on a separate heap without compaction. Such large objects are, however, rare in embedded systems according to our experience.

4.5 Overhead

Here, we take a closer look at the costs for memory management when the proposed GC algorithm and scheduling strategy are used.

4.5.1 High-priority processes, service time

The overhead for GC activities for a high-priority process consists of tight and bounded delays for memory allocation, pointer dereferencing, and pointer assignment.

Pointer access

A simple read barrier is required by the underlying GC algorithm: Pointer access is done indirectly through a forwarding pointer in the object. The overhead for this is 1 machine instruction. In addition, processor interrupts must be disabled during the pointer dereferencing and object access.

Pointer assignment

The worst-case overhead of the write barrier implied by our scheme, for a high-priority process, is in the order of 20 machine instructions. The write barrier guarding pointer assignments might require that an object is marked for later evacuation, but the actual copying is not performed.

Memory allocation

The proposed scheme guarantees that memory allocations by the high-priority process can be performed without GC or memory initialization work. Allocation involves moving a heap pointer and initializing object header information. The overhead consists of about 10 machine instructions.

4.5.2 Low-priority processes, service time

The garbage collector inflicts somewhat different overheads for the soft real-time part of a control system implemented by low-priority processes than for the high-priority processes.

Pointer access

Pointer access is always done indirectly via a forwarding pointer in the object. This incurs an overhead of 1 machine instruction. Disabling/enabling process switching during accessing costs another 2 instructions.

Pointer assignment

Pointer assignment is performed in the same way for both high-priority and low-priority processes. Thus, the overhead is the same, in the order of 20 machine instructions.

Memory allocation

When a low-priority process requests memory, the garbage collector is invoked in order to ensure that the current GC ratio does not drop too low. In the worst case an amount of GC work proportional to the size of the new object is performed. The exact worst-case cost of an allocation operation depends on maximum object size, total heap size, maximum amount of live objects, and the maximum amount of GC work that has to be performed during one GC cycle.

4.5.3 Summary of worst-case mutator overhead

The overhead for primitive pointer operations, described in Section 4.5.1 and 4.5.2, directly affects the response times of the processes of the application program (the mutator). As we have noted, high-priority processes are treated differently than low-priority processes, which yields different overhead depending on the priority. Table 4.1 summarizes and compares the overhead for high and low-priority processes.

	High-priority process	*Low-priority process*
Pointer access	indirect via forwarding pointer, one extra machine instruction[a]	indirect via forwarding pointer, one extra machine instruction[a]
Pointer assignment	write barrier, ≈20 machine instructions	write barrier, ≈20 machine instructions
Allocation	allocate object, ≈ 10 machine instructions	perform GC work, allocate object and zero contents, time proportional to the size of the object

Table 4.1 Summary of overhead for primitive pointer and memory management operations.

a.Access to pointers and object contents must be performed as atomic operations, adding overhead for disabling/enabling interrupts.

4.5.4 High-priority processes, latency

The worst-case latency consists of two parts: the time needed by the garbage collector to finish an on-going piece of atomic GC work, and the time needed by the process scheduler to perform the context switch. The time necessary to complete an atomic operation is short and bounded. The time needed by the process scheduler to perform the context switch often stands for the major part of the latency.[1]

1.A context switch takes about 64 µs in the real-time kernel used to test our garbage collector (25 MHz Motorola 68040 microprocessor), whereas the longest atomic operation encountered in our prototype garbage collector is 38 µs. See also Chapter 6.

4.5.5 Cleaning up after the high-priority processes

An issue to consider is how much time will be needed for GC work in between the activations of the high-priority processes. The amount of work needed depends on how much memory was allocated by the high-priority processes. Since high-priority processes in a control system are written to be fast and small they will in most cases allocate little or no memory. The amount of work that has to be performed will thus be small in most cases.

A semispace flip may also have to be performed between executions of the high-priority processes. The flip itself adds very little to the overhead since it does not involve more than changing the meaning of fromspace and tospace. Most incremental copying GC algorithms prescribe that the flip should be immediately followed by scanning the entire root pointer set and evacuating all the objects referenced by root pointers. The flip and the subsequent scanning of the root pointer set is to be performed as one atomic operation. Such an approach is not feasible in hard real-time systems, since it would give rise to a too long atomic operation. We therefore scan the root pointer set incrementally, making the flip very cheap. The flip can be implemented by some 10 machine instructions.

4.5.6 Additional work for the programmer

The proposed garbage collector requires some information about the application program in general, and the high-priority processes in particular, in order to schedule the GC work in such a way that it does not interrupt the high-priority processes. The programmer must estimate how much memory must be reserved for the high-priority processes, M_{HP}. In order to do this accurately, knowledge about the process set is required. The *periods* and *worst-case execution times* for each high-priority process along with its *worst-case allocation need* during one invocation must be known. Section 4.8 describes how to calculate M_{HP} and how to verify that a given process set is schedulable.

The programmer must also specify the *maximum amount of memory occupied by live objects*, which is used to calculate the minimum GC ratio.

4.6 Degradation during system overload

The scheduling principle proposed in this thesis assumes that enough CPU time remains for garbage collection after executing the high-priority processes to keep up with the allocation need of these processes. In Section 4.8 we discuss how to analyse a given process set and determine whether the garbage collector will get enough execution time. In practice, however, this does not give us a 100% guarantee that the garbage collector will keep up. Several factors can cause the result of the analysis to be erroneous. For example, the worst-case estimations of execution

times and allocation needs required as input to the analysis might be too optimistic, or the process model used in the analysis might not be completely adequate for the computer system in question. Therefore, despite a correct theory, our system must be able to handle overload situations in a reasonable way. It should provide *graceful degradation* in the presence of overload, i.e. the system should continue to function reasonable even when overloaded, even if all deadlines cannot be met. The least important parts of the system should be disturbed first. Parts with higher and higher priority should be disturbed as the overload increases.

If we do not introduce a mechanism for detecting overload and dealing with it, we quickly get into trouble when overload occurs. Suppose that all available CPU time is assigned to executing high-priority processes, leaving no time for garbage collection or initialization of memory. New objects would then continuously be allocated in tospace, eventually filling it up completely, with no memory left for allocating new objects. To alleviate the problem we would have to complete the evacuation of fromspace and perform a flip. This is not possible, however, since no free store would be available in tospace to hold the evacuated objects. A deadlock would result. In addition, allocating non-initialized memory causes several types of problems. Immediate problems occur if the application program relies on the fields of a newly allocated object having well-defined initial values. Other problems might arise when the collector is eventually resumed. Random bit-patterns might be mistaken for valid pointers or newly allocated objects might be overwritten in an effort to initialize more memory.

The above scenario could be avoided if the memory manager checked the initialization and evacuation status in connection with every allocation operation invoked by a high-priority process. Initialization work must be triggered immediately if the amount of pre-initialized memory is smaller than the requested memory block. GC work might also be required in connection with the allocation.

As was shown in Section 4.4.4, there is no risk of a deadlock as long as the amount of newly allocated memory in tospace, A, is less than F_{min}. If an allocation request would cause A to get larger then F_{min}, we could start the garbage collector, finishing the current GC cycle by evacuating the remaining live objects in fromspace. However, this would not provide a very graceful degradation. In the worst-case, a GC cycle would just have been commenced when the overload started. We could thus be forced to perform a complete GC cycle in order to avoid deadlock, causing a delay with a duration of perhaps several seconds. Even though this is preferable to a complete deadlock, it is not acceptable in a control environment.

Graceful degradation can be achieved by detecting overload early. Then, small amounts of GC work can be performed in an sequential fashion to keep the garbage collector from getting behind more than allowed. As was described in Section 4.4.4, the goal of the garbage collector is to make sure that the current GC ratio is equal to, or higher than, the minimum GC ratio. That is, the collection state should be described by a point somewhere on, or to the left of, the continuous line in Figure 4.9. Allocation requests made by high-priority processes are allowed to tem-

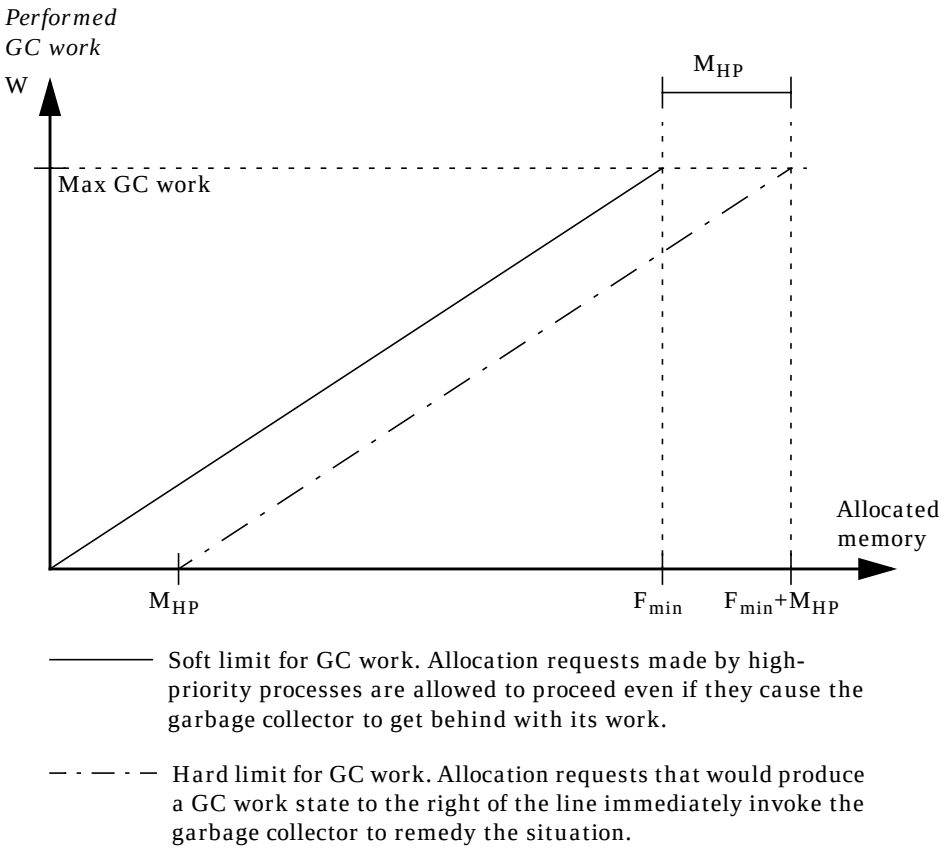


Figure 4.9 Handling overload. As overload causes the garbage collector to get behind with its work more than acceptable, allocation requests made by high-priority processes start to behave like allocation requests made by low-priority processes, performing an increasing amount of GC work with increasing overload.

porarily move the current collection state to the right side of the continuous line. If overload is not present, the collection state will never be more than M_{HP} units away from the line, i.e. it will stay to the left of the dashed line in the figure. Thus, if an allocation request would cause the collector state to end up somewhere to the right of the dashed line, the garbage collector is started to remedy the situation. More formally, the garbage collector should ensure that the following invariant is satisfied:

$$W \geq (A - M_{HP}) \cdot GCR_{min} \tag{4.7}$$

The modified allocation operation is described by the following pseudo code:

```
FUNCTION HP_Allocate(ObjectSize);
   WHILE GC work is motivated due to overload DO
      Perform an increment of GC work;
   END;
   WHILE P>T-ObjectSize DO
     P := P-1;
     MEMORY(P) := 0;
   END;
   T := T - ObjectSize;
   Initialize object header information;
   RETURN T;
END
```

In the presence of overload, high-priority processes will start to behave like low-priority processes in the sense that initialization and GC work are interleaved with the execution of the processes. The higher the overload, the more work is performed in this way. In extreme cases of overload, all initialization and GC work will be performed interleaved with the execution of the high-priority processes. If the overload continues to increase, the processes with lowest priority among the high-priority processes will be suspended first. The highest-priority processes, together with the initialization and GC work necessary to support their allocation need, will continue to execute.

4.7 Measuring garbage collection work

When defining rules for scheduling GC work we have used the term *work* without defining it more precisely. This section discusses various possible metrics for the work of an incremental copying garbage collector such as the one described in Section 4.3.

4.7.1 Work metrics

The ideal work metric would be to use the amount of real time actually used for performing garbage collection. It is ultimately the time spent on garbage collection we want to distribute as evenly as possible. However, this is not a very practical metric in reality since we typically do not have any means to measure time with high enough precision (microsecond resolution). Furthermore, it is also non-trivial to accurately estimate the worst-case amount of time required for a GC cycle. We thus require a more practical metric for the GC work. Some kind of approximation strategy have to be used.

A good metric has to correspond well with the ideal metric based on real time. A good metric should also be simple and easy to compute from the data available to the garbage collector. Using the metric, i.e. computing the amount of performed

work, should be cheap. Unfortunately, these requirements are contradictory. Consequently, we have to construct a metric that constitutes a reasonable compromise between the different demands.

4.7.2 The evacuation pointer metric

For a copying algorithm, a very simple metric can be obtained by using the amount of evacuated memory as a measure of performed GC work. The amount of evacuated memory is given by the position of the evacuation pointer, B in Figure 4.10, relative to the start of tospace. Let us denote the relative position ΔB. The maximum amount of work that may be required is equal to the maximum amount of memory that might be evacuated from fromspace, E_{max}. This is equal to the maximum amount of simultaneously live memory. We summarize:

$$\begin{aligned} W &= \Delta B \\ W_{max} &= E_{max} \end{aligned} \tag{4.8}$$

This is a metric used in many systems and research papers. It is very simple and easy to calculate, but it does have a serious disadvantage. The problem is that the correlation between the metric and actual time spent on garbage collection is bad. Each allocation request will cause the garbage collector to perform an amount of work proportional to the amount of requested memory. This means that an amount of memory will be evacuated proportional to the size of the requested memory block. The actual time it takes to evacuate a certain amount of memory varies greatly. In the worst case, the garbage collector will have to scan the contents of a large number of objects in order to find an unevacuated object to copy.

Imagine the situation in Figure 4.11, in which all live objects have been evacuated except for one. The only remaining pointer to the unevacuated object is located in the most recently evacuated object. The garbage collector will have to scan all the evacuated objects in tospace before the pointer to the unevacuated object is found and the object can be evacuated. Since the metric only uses the evacuation pointer to determine the amount of performed work, the garbage collector will not detect any change in the amount of performed work until the unevacuated object

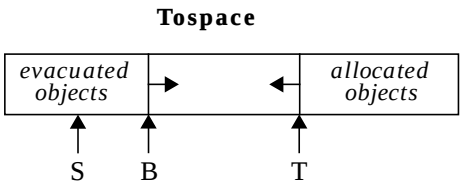


Figure 4.10 Structure of tospace of a typical copying GC algorithm.

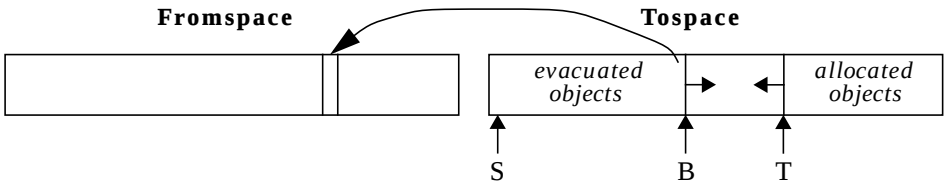


Figure 4.11 Worst-case situation for the evacuation pointer metric. Virtually all previously evacuated objects must be scanned before an unevacuated object is found and the metric indicates an increased amount of performed work.

is found and copied. E_{max} bytes must be scanned in the worst case, which may take a significant time.

If we schedule the GC work in connection with allocation requests, i.e. sequentially, any allocation request could result in a long delay as described above. This is clearly unacceptable in hard real-time systems. If we use semi-concurrent scheduling of the GC work as proposed in this thesis, on the other hand, the problem is reduced. Since no GC work is performed when high-priority processes execute, and since GC work is interruptible, only the low-priority processes will be delayed. The critical high-priority processes can continue to execute without delays. An exception to this is if we use the method described in Section 4.6 to achieve graceful degradation in case of system overload. Then, a bad metric might erroneously cause the garbage collector to believe that the system is overloaded. GC work would then be performed interleaved with the execution of the high-priority processes causing potentially long pauses.

4.7.3 Improving the evacuation pointer metric

The evacuation pointer metric takes only the actual evacuation of objects into account in its approximation of the ideal metric. However, the garbage collector spends time on other activities as well. Scanning objects for pointers into fromspace takes time, as does scanning root pointers. Memory initialization required at the end of the GC cycle (described in Section 4.4.1) is not modelled either. The metric can be improved by taking these activities into account as well. This is the metric we chose for the prototype garbage collector implementation described in Chapter 5.

In our modified model of the garbage collector, a GC cycle consists of four activities: scanning roots and evacuated objects, evacuation, and memory initialization. Denote the number of scanned root pointers *roots* and the maximum amount of simultaneously existing roots $roots_{max}$. The latter parameter must be provided by the programmer or compiler. The difference between the value of the scan pointer, S, and the initial value of S is denoted ΔS. Finally, let ΔP denote the amount of initialized memory in fromspace. The maximum amount of memory that has to be

initialized is equal to M_{HP}. The current amount of work, and the maximum amount of work that may be required during a GC cycle can now be calculated:

$$\begin{aligned} W &= \alpha \cdot roots + \beta \cdot \Delta S + \Delta B + \gamma \cdot \Delta P \\ W_{max} &= \alpha \cdot roots_{max} + \beta \cdot E_{max} + E_{max} + \gamma \cdot M_{HP} \end{aligned} \tag{4.9}$$

The maximum amount of memory to scan is equal to the maximum amount of memory to evacuate, which is in turn denoted E_{max}. This is the reason E_{max} occurs twice in (4.9). The actual time it takes to scan a root pointer is not the same it takes to evacuate a byte of memory. The cost of scanning an amount of memory is not the same as evacuating the same amount of memory or initializing a byte of memory either. The three factors α, β, and γ in (4.9) are used to compensate for this. They are supposed to be chosen in such a way that the metric deviates as little as possible from the ideal metric. This metric thus requires an amount of tuning to perform well (see also Section 5.2.6). It is conceivable that this tuning can be done automatically after a short test run when the system is started.

It is important to remember that the model is still only a simplification of the real case. For example, we assume that the time required to scan an object is proportional to the size of the object, which is not entirely true. In reality, the time depends on the number of pointers within the object, not its size. The metric will thus still deviate from the ideal one, but the worst-case deviation will be considerably smaller than for the evacuation pointer metric.

4.7.4 A fine-grained metric

Bengtsson presents a theoretically very sound strategy in his thesis [Ben90]. He suggests calculating the cost of every type of primitive operation performed by the garbage collector, such as moving one memory cell, traverse a pointer, performing a flip, etc. The costs are given in *GC units*. When the application is written and the, from the garbage collection point of view, worst-case state of the applications data structures is known, it is possible to calculate the worst-case amount of GC units needed for a GC cycle. This is done by adding up the costs of all the individual steps that must be performed. Progress is calculated by the collector incrementing a *work counter* each time it performs a primitive operation.

Bengtsson's approach gives very well-defined worst-case delays but suffers from three practical problems. Firstly, it can be tricky to achieve good estimates of the costs for each primitive operation. Secondly, calculating a good estimate of the worst-case amount of total work needed calls for quite detailed knowledge of the design and behaviour of the application. Thirdly, and probably most important, the housekeeping necessary to update the work counter can be very expensive.

4.7.5 Hardware support

An interesting approach to achieving a good metric would be to equip the computer system with a high-resolution clock. This would require additional hardware, but a simple counter incremented by a high-frequency clock would suffice. Here, the currently performed work, *W*, is equal to the cumulative time spent on garbage collection, whereas W_{max} is the maximum amount of time required for garbage collection during one GC cycle. Implementing a garbage collector based on this metric would probably require some extra support from the process scheduler in order to measure the execution time of the garbage collector correctly, since the scheduler is the only component that knows whether the garbage collector executes or is suspended.

Adding a hardware clock might be easy in open system architectures, such as VME-based systems. In many embedded systems it might be more difficult, however. It might be important to keep the hardware at a minimum, maybe for cost reasons or to minimize power consumption. Consequently, we do not believe an extra hardware clock is a generally acceptable solution.

4.7.6 Impact of imperfect metrics

The metric used to describe performed GC work will always deviate somewhat from the ideal metric. This will have a direct impact on the worst-case response time of a process if the work of the garbage collector is scheduled in connection with allocations, as it is in Baker's original incremental garbage collector [Bak78] and many other collectors. If we schedule the work according to our semi-concurrent strategy, on the other hand, the response time for a high-priority process will not be affected, since no GC work is performed when such processes execute. The worst-case response time of the GC process, see Section 4.8.2, will, however, be affected, which must be taken into account when doing schedulability analysis. The response times of low-priority processes will also be affected. Even so, a garbage collector using semi-concurrent scheduling will be less sensitive to imperfect metrics, since important control processes are not delayed.

4.7.7 Conclusions

Metrics for GC work can be defined in several ways, each with different advantages and disadvantages. Which one to choose depends on the requirements of the application with respect to worst-case delays and overall performance. Using the semi-concurrent scheduling strategy described in this thesis makes it possible to use a simpler method to calculate GC work without violating the real-time demands of the application.

4.8 Scheduling analysis

Predictability is an important factor when designing safety-critical systems. It is important to be able to derive the worst-case costs of individual operations. It is also important to be able to analyse the system as a whole and make sure that all hard deadlines are always met. Finding timing errors by testing the system in a real environment is sometimes not an option since a failure could be very costly and cause serious damage. In this section we describe how to perform a priori analysis.

In the analysis we assume that the application is constructed as a set of periodically executing high-priority processes which must be guaranteed to meet their deadlines in every situation. In addition, a number of low-priority processes exist which have soft real-time requirements, i.e occasionally missing a deadline can be accepted. We further assume that a fixed-priority scheduling strategy is used, such as rate monotonic scheduling or deadline monotonic scheduling. It should be added, however, that priority inheritance protocols do work with the analysis. We will also examine how non-periodic processes can be incorporated in the analysis.

The goal of the analysis presented in this section is to make it possible for a developer to determine that the high-priority processes of an application will always meet their deadlines and that there will always be enough time for the garbage collector to make sure that the high-priority processes do not run out of memory. The analysis is a two-step process. First, we use traditional scheduling analysis, more specifically rate monotonic analysis, to determine whether the high-priority processes are schedulable in every situation. After that has been proven, the analysis goes on by testing whether the remaining CPU time is enough for the garbage collector to keep up with the allocation requests of the high-priority processes. The notation introduced in the presentation of the scheduling analysis is summarized in Table 4.2.

4.8.1 Schedulability of the high-priority processes

The first step in verifying that a set of high-priority processes including its corresponding GC work is schedulable consists of determining the schedulability of the high-priority processes alone. This can be done in various ways, depending on the scheduling strategy employed.

One way of testing the schedulability of a set of processes scheduled according to the rate monotonic principle, is to study the processor utilization, denoted U. Assume a system consisting of N high-priority processes $\tau_1..\tau_N$, with periods $T_1..T_N$

τ_i	process number i
T_i	period for process τ_i
C_i	worst-case execution time for one invocation of process τ_i
D_i	deadline for process τ_i relative to the scheduled invocation time
R_i	worst-case response time for process τ_i
A_i	worst-case memory allocation need during one invocation of process τ_i
G_i	worst-case time required for garbage collection after one invocation of process τ_i including the time required for memory initialization
d_i	worst-case time process τ_i can be blocked by low-priority processes
c_i	worst-case time required for garbage collection during one invocation of process τ_i due to the actions of low-priority processes blocking τ_i.
B_i	worst-case blocking time for process τ_i including direct blocking and push-through blocking
N	number of high-priority processes in the system
U	processor utilization
R_{GC}	worst-case response time for the GC process
C_{GC}	required execution time for the GC process
M_{HP}	amount of memory that has to be reserved in tospace in order to guarantee that it always will be possible to meet a memory allocation request from a high-priority process

Table 4.2 Scheduling analysis notation.

and worst-case execution times $C_1..C_N$ respectively. Then the processor utilization is defined by:

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \tag{4.10}$$

Liu and Layland derived a utilization bound that can be used to test the schedulability of the processes [LL73]. Their test states that if the processor utilization is

less than, or equal to, the utilization bound, the system is schedulable. Their condition for schedulability is formulated mathematically as:

$$U \leq N \cdot \left(2^{\frac{1}{N}} - 1 \right) \tag{4.11}$$

This is a sufficient but not necessary condition, however. That is, a system might be schedulable even if the processor utilization is higher than Liu and Layland's utilization bound. When the condition is not met, some other method has to be used to check schedulability. Another problem with the schedulability test of Liu and Layland is that it assumes a very strict process model. All processes must be periodical with deadlines equal to their periods. Blocking is not allowed, which is very unpractical since it makes communication and synchronization between processes very difficult.

Joseph and Pandya presented in 1986 a method for exact analysis of process schedulability that can be applied to all fixed-priority scheduling strategies [JP86]. The basic idea is to calculate the worst-case response time for each process, R_i,and compare it with the deadline for the process, D_i. If R_i is smaller than or equal to D_i for each process, the process set is schedulable. If we for a moment assume that no blocking occurs, the worst-case response time of a process is equal to the worst-case execution time of the process, C_i, plus the time the process has to wait for higher-priority processes to execute. The time a process might have to wait for each higher priority process is equal to the worst-case execution time for the higher priority process multiplied by the number of times the process with higher priority can be invoked during the response time of the process we are analysing. This can formally be written as:[1]

$$R_i = C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right) \tag{4.12}$$

R_i cannot be directly calculated from (4.12) since R_i is found on both sides of the equality. However, as shown in [JP86], it can be calculated iteratively:

$$\left.\begin{aligned} R_i^0 &= 0 \\ R_i^{n+1} &= C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \right) \end{aligned}\right\} \tag{4.13}$$

1.We use $\lceil \rceil$ to denote the ceiling function, i.e. the smallest integer that is equal to, or larger than, the function argument.

When (4.13) converges, we have found the worst-case response time.

Much work has been devoted to generalizing the analysis of Liu and Layland to handle more complex process models. An example is the generalized rate monotonic analysis [SRL94]. One important extension is the ability to handle blocking. Processes may be blocked in two ways. A process can be blocked directly by a lower-priority process that holds a resource that the process requests. It can also be blocked indirectly if a priority inheritance protocol is used. In the latter case, a process is first preempted by a higher-priority process which is then in turn directly blocked by a lower-priority process sharing a resource with the high-priority process. The low-priority process inherits a higher priority and effectively blocks both the processes with higher priority. This is called push-through blocking. If we can calculate the worst-case blocking time for each process, B_i, we can modify (4.12), which yields:

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right) \tag{4.14}$$

The blocking time, B_i, depends on which priority inheritance protocol is used, since this affects the worst-case push-through blocking.

4.8.2 Schedulability of the garbage collector

Assuming that the high-priority processes of a system have been determined to be schedulable, let us now verify that the GC work motivated by the actions of the high-priority processes is schedulable as well.

Consider the worst-case scheduling situation, in which all high-priority processes are released simultaneously. As shown in [LL73], if we can schedule this situation we can also schedule all other situations. Each process, τ_i, executes for a duration equal to its worst-case execution need, C_i, and performs memory management related actions that requires a worst-case GC/memory initialization work of G_i to be performed. The worst-case response time of the garbage collector, R_{GC}, can now be defined as the time from the high-priority processes were released until no more GC work is left to be performed. R_{GC} can be calculated in a similar way to how the response times were calculated for the high-priority processes in Section 4.8.1:

$$R_{GC} = C_{GC} + \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot C_i \right) \tag{4.15}$$

Equation (4.15) contains C_{GC}, which in our case depends on the actions of the high-priority processes. In other words, it is not fixed as the execution times for the high-priority processes were in Section 4.8.1. For each invocation of a high-priority proc-

ess τ_i during R_{GC}, the required GC work amounts to G_i. The total GC work during R_{GC} will therefore be:

$$C_{GC} = \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot G_i \right) \tag{4.16}$$

Applying (4.16) to (4.15) yields:

$$R_{GC} = \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot (C_i + G_i) \right) \tag{4.17}$$

We find that R_{GC} is found on both the left side and the right side of the equality. The smallest non-zero value of R_{GC} that satisfies (4.17) can be found using the recursive formula:

$$\left.\begin{aligned} R_{GC}^{0} &= \sum_{i=1}^{N} C_i \\ R_{GC}^{n+1} &= \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}^{n}}{T_i} \right\rceil \cdot (C_i + G_i) \right) \end{aligned}\right\} \tag{4.18}$$

It should be noted that we cannot use 0 (zero) as the first approximation of R_{GC}, as we did in Section 4.8.1 when calculating the worst-case response time of a process. This is because 0 is a trivial solution to (4.17), whereas the solution we want is the first positive, non-zero solution. Clearly, R_{GC} cannot be smaller than the sum of the worst-case execution times for the high-priority processes since all processes are released simultaneously in the worst case and the garbage collector has lower priority than these processes. The garbage collector will thus not be assigned processor time until each high-priority process has run at least once. If the worst-case execution times of the processes are small compared with their periods and the processes do not perform any action that motivates GC work, the response time of the garbage collector will be equal to the sum of the worst-case execution times of the processes. Any value higher than 0 and equal to or lower than the sum of the worst-case execution times of the processes can be chosen as the initial value in the iteration. It is however convenient to choose as large a value as possible, that is still easy to calculate, in order to avoid unnecessary iterations.

If the GC work is schedulable, (4.18) will converge. If the GC work is *not* schedulable, (4.18) will *not* converge since no solution exists. It is easy to detect that (4.18) has converged. This happens when two consecutive values of R_{GC} are found to be equal. The value of R_{GC} that the formula converges towards is the worst-case response time of the garbage collector. But how do we detect that the formula does

not converge? The answer to this is that we can calculate a largest possible value for R_{GC}. If one of the steps in the iterative process of calculating R_{GC} yields a value larger than the maximum possible response time, we can deduce that the iteration will not converge.

Theorem. The maximum possible response time for the garbage collector is the *least common multiple of the periods of the high-priority processes*, denoted $lcm(T_1..T_N)$.

If we, for example, have a system with two high-priority processes with periods of 10 and 14 milliseconds respectively, the response time of the garbage collector must be less than or equal to $lcm(10,14) = 70$ milliseconds.

Proof. Assume that all the high-priority processes are released simultaneously at time t. This is the worst-case scheduling situation, as shown in [LL73]. The processes will execute with different periods forming a scheduling pattern. Sooner or later they will all again be scheduled to run simultaneously, after which the scheduling pattern will repeat itself. This happens at time $t+lcm(T_1..T_N)$. Thus, if there was not enough time in the time slot $t..t+lcm(T_1..T_N)$ to complete the GC work in progress, there will in the worst case not be enough time in the next time slot either, and so on. The amount of needed garbage collection will continue to accumulate. The response time of the garbage collector must therefore be less than, or equal to, $lcm(T_1..T_N)$.

4.8.3 Memory reserved for high-priority process usage

The copying GC algorithm we have used to illustrate our scheduling strategy may experience a deadlock if the evacuation of live objects from fromspace does not keep up with the allocation of new objects. When low-priority processes allocate memory, they make sure that enough GC work has been performed before they actually allocate the new object, which guarantees that the low-priority processes do not cause any deadlock. High-priority processes, on the other hand, allocate memory before the corresponding GC work is performed. This is a potentially dangerous situation if the high-priority processes are invoked shortly before a semi-space flip is due, since there might not be enough memory left in tospace to hold both the new objects and the live objects that have not yet been evacuated from fromspace. The solution to this problem is to schedule the GC work, and the semi-space flip, in such a way that enough memory remains in tospace for evacuation of live object even if high-priority processes are invoked immediately before the flip. This can be viewed as reserving an amount of memory for allocation by high-priority processes.

How can the amount of memory that has to be reserved in tospace for high-priority allocation, M_{HP}, be determined? A worst-case estimation can be obtained by assuming that all high-priority processes are released immediately before a flip is to be performed. We furthermore assume that the flip cannot be performed within R_{GC} time units after the invocation of the high-priority processes. We must then reserve enough memory in tospace to hold all objects allocated during R_{GC} time

units. This amounts to the sum of the worst-case allocation needs for all the high-priority process invocations during R_{GC} time units. The allocation need for a process can be calculated by multiplying the worst-case allocation need during one invocation with the number of times the process might be invoked during a time span of R_{GC}. We get:

$$M_{HP} = \sum_{i=1}^{N} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i \tag{4.19}$$

M_{HP} is also the amount of memory that the garbage collector must keep initialized and ready for allocation by high-priority processes. During any time interval of length R_{GC}, the high-priority processes might allocate M_{HP} bytes of memory. By including the time required for initialization in G_i, a system which is schedulable according to the analysis presented in this chapter will have time to initialize M_{HP} bytes of memory during any interval of length R_{GC}, which will provide enough initialized memory for the subsequent interval, et cetera.

4.8.4 Scheduling analysis example

To illustrate the scheduling analysis we will study two example sets of high-priority processes and determine whether they are schedulable or not.

Example 1 - a schedulable set

Consider the set of three high-priority processes whose process attributes are found in Table 4.3. We assume that rate monotonic scheduling has been used to assign priorities to the processes, giving τ_1 the highest priority and τ_3 the lowest. We also assume that the deadline for a process, D_i, is equal to the period of the process, T_i.

Process	T_i *(ms)*	C_i *(ms)*	A_i *(bytes)*	G_i *(ms)*
τ_1	10	3	72	1
τ_2	50	9	302	5
τ_3	95	21	256	4

Table 4.3 Process parameters of a schedulable set.

Rate monotonic analysis, as described in Section 4.8.1, gives the following worst-case response times for the three processes:

$$R_1 = 3,\ R_2 = 15,\ R_3 = 45$$

We observe that $R_1 \leq D_i$ for each process ($D_i = T_i$). Thus, the three high-priority processes are indeed schedulable.

The next step is to ensure that there are enough time to perform the necessary GC work. Equation (4.18) on page 77 is used to calculate the worst-case response time of the garbage collector, R_{GC}:

$$R_{GC}^0 = 3 + 9 + 21 = 33$$

$$R_{GC}^1 = \left\lceil \frac{33}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{33}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{33}{95} \right\rceil \cdot (21+4) = 55$$

$$R_{GC}^2 = \left\lceil \frac{55}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{55}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{55}{95} \right\rceil \cdot (21+4) = 77$$

$$R_{GC}^3 = \left\lceil \frac{77}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{77}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{77}{95} \right\rceil \cdot (21+4) = 85$$

$$R_{GC}^4 = \left\lceil \frac{85}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{85}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{85}{95} \right\rceil \cdot (21+4) = 89$$

$$R_{GC}^5 = \left\lceil \frac{89}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{89}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{89}{95} \right\rceil \cdot (21+4) = 89$$

The recursion has converged, which tells us that the system is schedulable. The worst-case response time of the garbage collector will be 89 milliseconds. The system, including garbage collection, is thus schedulable. We observe that the recursion converged towards a value less than $lcm(10,50,95) = 950$, which is the maximum possible response time for the garbage collector.

Having determined R_{GC} we can now also calculate the minimum amount of memory that has to be reserved in tospace for allocation requests by the high-priority processes, M_{HP}. Equation (4.19) on page 79 yields:

$$M_{HP} = \left\lceil \frac{89}{10} \right\rceil \cdot 72 + \left\lceil \frac{89}{50} \right\rceil \cdot 302 + \left\lceil \frac{89}{95} \right\rceil \cdot 256 = 1508$$

We conclude that the given system is schedulable and that 1508 bytes should be reserved in tospace for allocations performed by the high-priority processes in order to ensure that the GC algorithm never deadlocks. The less time required for execution of the high-priority processes, the shorter the response time for the garbage collector will be. Shorter response time for the garbage collector in turn

makes the amount of memory, M_{HP}, reserved for allocation by high-priority processes smaller.

Example 2 - a non-schedulable set

Let us study the set of high-priority processes described by Table 4.4. We assume once more that rate monotonic scheduling has been used to assign priorities to the processes and that $D_i = T_i$ for each process. Is this set of processes and the corresponding GC work schedulable?

Process	T_i *(ms)*	C_i *(ms)*	G_i *(ms)*	A_i *(bytes)*
τ_1	10	3	1	72
τ_2	50	9	5	302
τ_3	75	21	4	256

Table 4.4 Process parameters of a non-schedulable set.

We begin by investigating the schedulability of the processes using rate monotonic analysis. This yields:

$$R_1 = 3,\ R_2 = 15,\ R_3 = 45$$

The processes alone are thus schedulable since $R_i \leq D_i$ for each process.

Next, it must be determined whether the GC work is schedulable or not. As discussed in Section 4.8.2, R_{GC} must be less than or equal to the least common multiple of the process periods ($R_{GC} \leq lcm(T_1..T_3)$) for the GC work to be schedulable. That is, (4.18) on page 77 must converge on a value less than or equal to $lcm(10,50,75) = 150$ for the set of processes to be schedulable. Let us investigate if this is the case:

$$R_{GC}^0 = 3 + 9 + 21 = 33$$

$$R_{GC}^1 = \left\lceil \frac{33}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{33}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{33}{75} \right\rceil \cdot (21+4) = 55$$

$$R_{GC}^2 = \left\lceil \frac{55}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{55}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{55}{75} \right\rceil \cdot (21+4) = 77$$

$$R_{GC}^3 = \left\lceil \frac{77}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{77}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{77}{75} \right\rceil \cdot (21+4) = 110$$

$$R_{GC}^{4} = \left\lceil \frac{110}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{110}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{110}{75} \right\rceil \cdot (21+4) = 136$$

$$R_{GC}^{5} = \left\lceil \frac{136}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{136}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{136}{75} \right\rceil \cdot (21+4) = 148$$

$$R_{GC}^{6} = \left\lceil \frac{148}{10} \right\rceil \cdot (3+1) + \left\lceil \frac{148}{50} \right\rceil \cdot (9+5) + \left\lceil \frac{148}{75} \right\rceil \cdot (21+4) = 152$$

We observe that R_{GC} does not converge on a value less than or equal to 150. Therefore, it is *not* possible to schedule this set of processes with associated garbage collection in the worst case.

4.8.5 The effect of blocking

In the analysis above we have assumed that the neither high-priority processes nor the high-priority GC process are blocked by other processes (excluding blocking caused by preemption). This is typically not the case in practical applications when it comes to the high-priority processes. High-priority processes can be blocked when using resources shared with other processes. We must therefore take blocking into consideration in order to correctly determine the schedulability of the high-priority processes. A detailed analysis will probably also take the costs of context switches and the process invocation jitter effect caused by atomic operations into account. Generalized rate monotonic analysis [SRL94] provides the means for the analysis.

The high-priority GC process, on the other hand, is independent from the execution of other processes. Therefore, blocking does not occur and the analysis of the schedulability of the GC work above is valid without modification. We might want to take the cost of context switches into account when calculating the effect of high-priority processes on the execution of the high-priority GC process, however.

The introduction of priority inheritance protocols changes the situation somewhat. Low-priority processes will now temporarily execute as high-priority processes, which affects the schedulability analysis of the high-priority GC process. In the following section, we will study different priority inheritance strategies and their impact on our analysis.

4.8.6 Priority inheritance protocols

To avoid blocking caused by priority inversion, priority inheritance protocols are employed. All of these protocols involve temporarily raising the priority of a process that has allocated a resource, which may cause a low-priority process to become a high-priority one until the resource is released. This must be taken into consid-

eration when analysing the schedulability of a system of processes. We will look at common priority inheritance protocols one by one and discuss what impact their use has on the scheduling analysis.

The basic inheritance protocol

The basic inheritance protocol states that whenever a process blocks because the resource it attempts to allocate is already allocated by a process with lower priority, the process currently holding the resource will inherit the priority of the blocked process. The priority of a process is thus raised if, and only if, it is blocking a higher priority process.

Blocking a high-priority process and raising the priority of the process causing the block to the priority level of the blocked process can be said to be equivalent to the case that the high-priority process is performing the work within the critical region of the process with lower priority. We can thus incorporate the basic inheritance protocol in our scheduling analysis by modifying (4.17) on page 77 slightly:

$$R_{GC} = \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot (C_i + d_i + G_i + g_i) \right) \tag{4.20}$$

We use d_i to denote the worst-case time a process spends performing work for lower-priority processes as described above. For each process, τ_i, we have to add d_i to the worst-case execution time of the process. While performing the work of a low-priority process, τ_i might take actions that motivate additional GC work to be performed. The additional worst-case time for GC work is denoted g_i, and must also be taken into account when calculating the response time of the garbage collector. It is worth to notice that it is only low-priority processes that influence d_i and g_i. The execution time and GC need of high-priority processes are already taken into account, even if they do block other high-priority processes.

To analyse a system utilizing the basic inheritance protocol, we must be able to determine the value of d_i and g_i. Any of the following two observations can be used to find an upper bound [SRL90]:

- Under the basic inheritance protocol, a process τ_i can be delayed at most once by each process with lower priority which share some resource with τ_i.
- Second, if m resources exist which can cause τ_i to block, then τ_i can be blocked at most m times, once by each resource.

By analysing the worst-case execution times and allocation need of the corresponding critical regions and adding them up, we can compute d_i and g_i.

The priority ceiling protocol

In the priority ceiling protocol, resources are assigned a *ceiling* priority. The ceiling priority is the priority of the highest-priority process that use the resource. A process is only allowed to allocate a resource if the priority of the process is strictly higher than the ceilings of all resources currently held by other processes. If this is not the case, the process is blocked. As in the basic inheritance protocol, whenever a process is blocked, the process that is causing the block, i.e holding a resource with a ceiling that is equal to or higher than the priority of the blocked process, inherits the priority of the blocked process.

Since a blocking process gets its priority raised to the priority of the process it is blocking, the situation is equivalent to the case that the blocked process is performing the execution of the lower-priority process while within the critical region. This is analogous to the case of using the basic inheritance protocol. We can thus use the same strategy to incorporate the priority ceiling protocol as we used to incorporate the basic inheritance protocol. Equation (4.20) is therefore valid also in this case.

The priority ceiling protocol has somewhat different blocking properties than the basic inheritance protocol. A process can be delayed at most once during each invocation by a lower priority process. Therefore, we only have to find the critical section in the one low-priority process that has the highest worst-case execution time in order to determine d_i. We determine g_i analogously.

The immediate inheritance protocol

The probably simplest priority inheritance protocol to implement is the immediate inheritance protocol. Resources are assigned ceiling priorities just like in the priority ceiling protocol. However, the priority of a process is raised not only when it is actually blocking another process. Instead, as soon as a process attempts to allocate a resource, the priority of the process is set to the maximum of its current priority and the ceiling priority of the resource. The priority of the process is reset to what it was before when the resource is later released.

Since the priority of a process is raised every time it allocates a resource (unless it already has a priority equal to or higher than the ceiling priority), a low-priority process sharing a resource with a high-priority process will become a high-priority process every time it allocates the resource. Our previous approach in which we regarded the work within the critical region as being performed by the blocked high-priority process does not work in this case, since there may not exist a blocked high-priority process. A different mathematical model is therefore required to make our scheduling analysis work with the immediate inheritance protocol.

The following observation helps us develop a working scheduling analysis for the immediate inheritance protocol: When a low-priority process allocates a resource and temporarily becomes a high-priority process, no other low-priority

process can possibly become a high-priority process since they, because of their lower priority, are not eligible for execution as long as the first process is executing with a raised priority, i.e as long as the resource is held. In order to determine if a system of high-priority processes and the corresponding GC work is schedulable, we study the worst-case scheduling situation, that is what happens when all high-priority processes are released at the same instant. One additional thing has to be taken into consideration when the immediate inheritance protocol is used, namely that one of the low-priority processes might have become a high-priority process by entering a critical region immediately before the regular high-priority processes are released. This can be modelled by adding a one-shot high-priority process to the analysis. The priority of the process must be the highest possible priority a low-priority process can possibly inherit. The execution time of the process must be the longest possible execution time a low-priority process might need to leave the critical region causing the raised priority. The worst-case GC work required by the process is determined similarly.

The first step in the scheduling analysis, i.e determining whether the high-priority processes are schedulable (see Section 4.8.1) is unchanged with the exception of the inclusion of the extra one-shot process (the one-shot property can be modelled by setting the period of the process to infinity). The response time for the garbage collector process can be calculated as follows, C_{LP} denotes the worst-case delay caused by the low-priority process and G_{LP} denotes the worst-case amount of GC work that is required by the low-priority process:

$$R_{GC} = C_{LP} + G_{LP} + \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot (C_i + G_i) \right) \tag{4.21}$$

4.9 Scheduling mark-sweep garbage collection

The scheduling principle proposed in this thesis is not restricted to copying algorithms for garbage collection. It can be applied to other types of algorithms as well. In this section we study how an incremental compacting mark-sweep algorithm can be adapted to our scheduling scheme.

4.9.1 The algorithm

The mark-sweep algorithm we have chosen was presented by Bengtsson in his licentiate thesis [Ben90]. His thesis contains a thorough description of the algorithm together with an analysis of its performance.

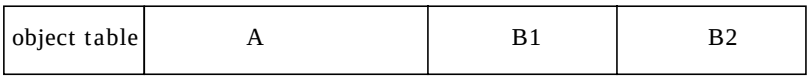


Figure 4.12 The heap structure of the incremental, compacting, mark-sweep collector.

Algorithm overview

The heap is divided into three areas called A, B1, and B2 respectively. In addition, a part of the memory is set aside for an object table containing pointers to the objects on the heap. All pointer referencing is done indirectly via the object table. Figure 4.12 illustrates the heap organization.

The algorithm works in odd and even cycles. Every other cycle is odd and every other cycle even. During odd cycles, new objects are allocated in area B1, whereas new objects are allocated in B2 during even cycles. As new objects are allocated in the current allocation area, the algorithm incrementally compacts the contents of the rest of the heap, i.e. areas A and B2 during odd cycles and A and B1 during even cycles. All live objects that are identified during the cycle are relocated to the lower end of area A. The area A must be large enough to hold all objects reachable at the beginning of a GC cycle, that is, at least as large as the maximum amount of simultaneously live memory. The allocation area that is to be used for allocation in the next cycle will thus be evacuated. When the allocation area, B1 or B2, fills up, new objects are allocated in A until A fills up as well. Then a new GC cycle commences. The algorithm is summarized in Figure 4.13

Object table

When an object is moved, all pointers to the object have to be updated. In order to do this in a short time, a pointer indirection scheme is used as illustrated in Figure 4.14. An object table contains an entry for each object on the heap. The entry in turn contains a pointer to the current location of the object. All object pointers point to an entry in the object table rather than directly to an object. When an object is moved, we only have to update one pointer, i.e. the entry in the object table. In order to quickly find the location of the entry when an object is to be moved, all objects contain a pointer to the corresponding object table entry.

Since every object requires a corresponding entry in the object table, the object table must have the same number of entries as the maximum number of objects that may exist in memory at any one time. Furthermore, as the garbage collector reclaims the memory occupied by an object, it must also reclaim the corresponding entry in the object table. We suggest that a free-list of unused entries is used. The cost for allocating a new entry from the list will be small and constant since we can

Collector

Marking phase:

- Mark unmarked objects referenced by roots and push references to them onto the collector stack.
- While the collector stack is not empty:

  Pop an object reference, X, from the collector stack.
  Mark all unmarked objects referenced by X and push references to them onto the collector stack.

Compaction phase:

- Traverse from low to high addresses the areas A and B2 (odd cycles) or areas A and B1 (even cycles):

  If an object is marked
  move it to the lowest free address in A and update the object table entry
  else
  reclaim its object table entry

Mutator

Pointer assignment:

- If the collector is in the marking phase and the referenced object is unmarked:

  Mark the object and push a reference to it onto the collector stack.

Pointer referencing:
Indirect through an object table.

Allocation:

- Allocate in B1 (odd cycles) or B2 (even cycles).
- When the collector cycle has finished and B1/B2 is full:

  Allocate in A as long as there are free memory, then start a new cycle.

Figure 4.13 Summary of Bengtsson's incremental, compacting, mark-sweep collector.

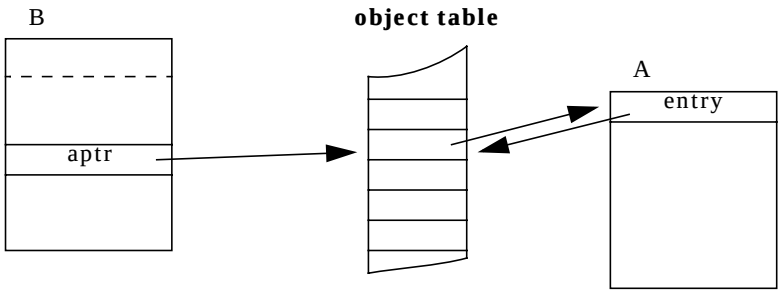


Figure 4.14 The pointer-indirection scheme of Bengtsson's incremental, compacting mark-sweep algorithm. All pointers point to an entry in an object table. The entry contains the only pointer to the referenced object. Every object contains a pointer back to the corresponding object table entry. In the figure, the pointer B.aptr points to object A via the object table.

always choose the first entry in the list. Similarly, the cost for inserting a freed entry into the list is also small and constant.

Collector stack

As the algorithm traverses the object graph during the marking phase, references to objects that are to be scanned are pushed onto a collector stack. Pointer reversal techniques [SW67,Tho76] cannot be used since they would temporarily invalidate the data structures and block the mutator for long periods of time. A trivial implementation of the collector stack would be to reserve a separate area in memory for the stack. In the worst case, the stack could contain a pointer to every object on the heap. Therefore, the size of the stack would have to be proportional to the maximum number of simultaneously live objects.

An alternative implementation of the collector stack is to reserve a word of memory in the header of each object. Objects on the collector stack are then linked together using pointers located in the object headers. The worst-case memory requirements of this scheme is equal to that of the approach using a separate stack. However, if we choose to use a separate stack, some GC information is still required in the objects themselves. The garbage collector must be able to determine whether an object is unmarked or not. Typically, one bit in the object header is used to store this information. If we implement the collector stack by pointers in the object headers, the word of memory used to store the pointer can also be used to hold information about the state of the object as well. An unmarked object could for example contain a null pointer. Any other bit pattern would indicate that the object is marked. This approach thus has the potential of reducing the memory requirements of the scheme somewhat.

4.9.2 Atomic operations

Some operations affecting the heap must be ensured to be executed atomically. Such operations either store temporary pointers in places unknown to the garbage collector or they temporarily cause the heap itself to be in an inconsistent state. We suggest that atomicity is achieved by disabling the processor interrupts in the same way as we proposed for the copying algorithm in Section 4.3.

Pointer dereferencing

The read barrier is implemented in the same way as for the copying algorithm, i.e. by one level of indirection. First, the current address of the target object is found by looking it up in the object table. Second, the retrieved address is used to access the object. If we would not disable the processor interrupts during these two steps, a context switch could potentially occur just after the mutator had retrieved the current address from the object table. The garbage collector could then be invoked and move the object being dereferenced by the mutator. The address fetched from the object table would no longer be correct.

Write barrier

The write barrier of Bengtsson's incremental mark-sweep algorithm is quite cheap. The barrier checks whether the collector is currently in the marking phase. If so, it checks whether the referenced object is unmarked, in which case it marks it and puts a reference to it on the collector stack. All of this can be performed by a small and bounded number of machine instructions. The cost of the write barrier should be comparable to, or cheaper than, the write barrier of our copying algorithm employing the lazy-evacuation scheme (Section 4.4.2). A pointer assignment including the write barrier is considered to be an atomic operation.

4.9.3 Interruptible garbage collection

If very short process latencies are required, relocating an object cannot be treated as an atomic operation. Large objects take too long time to copy. For the copying algorithm we solved this problem by using an optimistic approach to object copying, see Section 4.4.5. An object copying is started hoping that it will complete without being interrupted by a higher-priority process. If a context switch occurs, the garbage collector backs out of the on-going copying and retries when resumed later. This scheme will maintain the consistency of the data on the heap. At most one object copying has to be abandoned each time a context switch is performed, which bounds the extra overhead.

When using the copying algorithm, objects are always copied between separate memory areas, which guarantees that the source and destination areas never overlap. Object copying can therefore be interrupted at any time without violating the integrity of the old version of the object. If the basic mark-sweep algorithm is used, the source and destination areas may very well overlap. Overlap might result in that neither the old (source) nor the new (destination) copy is consistent; the new copy is not complete yet and the old copy has been partially overwritten by the new copy, see Figure 4.15.

In order to solve the problem with overlapping source and destination areas, the algorithm has to be modified in such a way that the source and destination areas of an object copying never overlap. Overlapping source and destination areas for object copying can be avoided in the following way, illustrated by Figure 4.16: The size of the area A is increased with the maximum object size and we alternate between moving objects towards lower and higher addresses in A. During odd GC cycles, live objects are slided towards lower addresses just as before, whereas they are slided towards higher addresses during even cycles. We also have to modify the allocation strategy slightly: When the current allocation area, B1 or B2, is filled up, we continue by allocating new objects in A. During odd cycles, new objects are allocated at the *lowest* possible address in A whereas they are allocated at the highest possible address during even cycles. We do not allow A to fill up completely before starting a new GC cycle, but a new cycle is started when an allocation would cause the amount of remaining free memory in A to become less than the maximum object size. An amount of unused space will consequently be left in A. When the next cycle starts, this space will be located at the end of A, towards which we will move all live objects. Thus, when the garbage collector enters the compaction (sweep) phase of the following cycle, we can guarantee that the distance between the source and destination addresses will be at least equal to the maximum objects size. All moves will therefore be non-overlapping.

The objects in A are traversed linearly in connection with the compaction phase. The traversion starts at the end of A towards which we will move the objects. During odd cycles we will traverse the objects from low addresses towards higher addresses whereas we during even cycles will traverse the objects in the opposite direction. If the header of each object contains the size of the object it is easy to

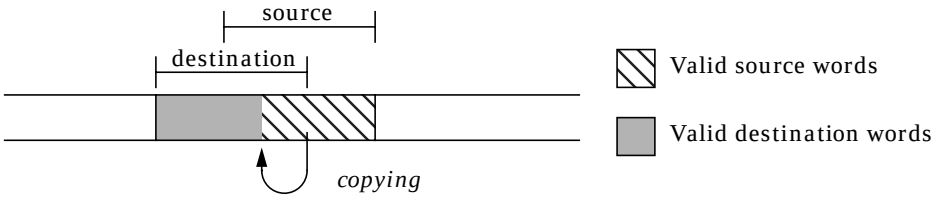


Figure 4.15 Overlapping source and destination areas during an object copying causing the object to get into a temporarily inconsistent state. Neither the source nor the destination area contains a complete object.

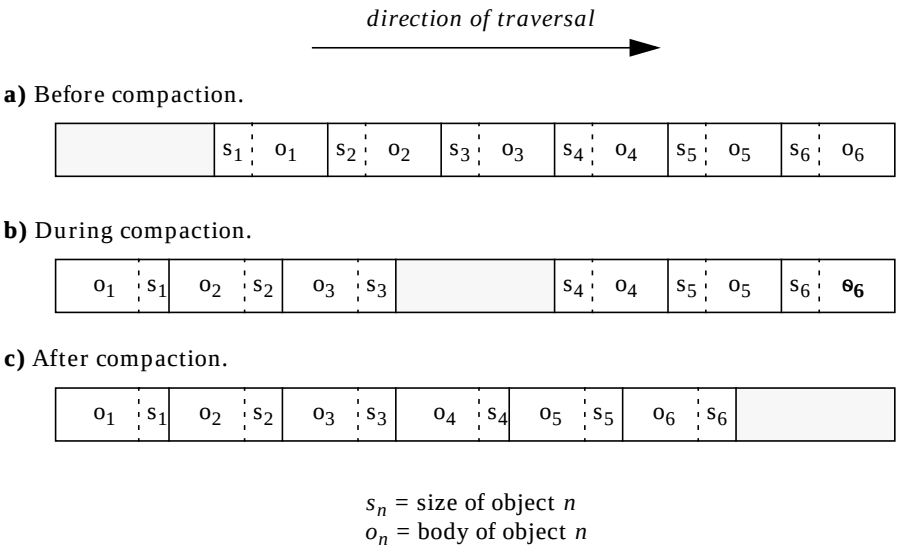


Figure 4.16 The layout of area A of our incremental mark-sweep algorithm during an *odd* GC cycle. Objects are slided towards low memory. As an object is moved, the objects size word is relocated to a position immediately after the object body in order to facilitate the traversal in the opposite direction of the next, even, cycle.
Even GC cycles are similar, but the objects are slided towards high memory and the size words are put before the object bodies.
The grey area must always be large enough to guarantee that no copying will involve overlapping source and destination areas. This makes it always possible to back out of an on-going copying.

traverse the objects in low-to-high order. To get from one object to the next we only have to add the size of the object to its address to get the address of the next object. High-to-low order traversion is not possible using this scheme. From an object we want to find the address of the following object (at a lower address). To find this address we need to subtract the size of the following object from the address of the current object, but we cannot find the size since we do not know the address of the following object.

Traversing the objects in A from high addresses towards lower addresses is possible if the information about the size of the objects is separated from the headers of the objects. During even cycles, as objects are relocated and later when new objects are allocated in A, we place a word of memory immediately in front of every object containing the size of the object. In the subsequent odd cycle, this information can be used to find the next object. During odd cycles, we instead place the size word immediately *after* the objects. In this way we will be able to find the start of the next object to traverse during the following even cycle, even though we traverse

the objects in a high-to-low order. The size of the object following the object that is currently being traversed can be found in the memory word just before the current object.

The described method of avoiding overlapping source and destination areas has both memory and time overhead. The memory overhead is equal to the maximum object size, which in most cases must be considered insignificant. The time overhead affects the average-case performance, which will be poorer. Alternating between moving the objects in A towards the lower and upper ends implies that every object in A will be moved once in every GC cycle. In the original formulation of the algorithm, old objects will tend to sediment at the lower end of A. Since the mortality of old objects is generally low [Ben90,Ung84], there is a good chance that the garbage collector can traverse a number of live objects without having to move them. They will already be at the right place. It is not until a dead object is encountered that the collector will have to start move objects. The worst-case behaviour is the same for both approaches, however. In the worst case, the object located at the start of A will be dead, in which case every object will have to moved, just as when we alternate between moving objects upwards and downwards in A.

4.9.4 Work scheduling

The work of the mark-sweep algorithm is scheduled in an analogous way to how it is scheduled for the copying algorithm we have discussed earlier, see Section 4.4.4. For low-priority processes, garbage collection is performed in connection with allocation request. An amount of garbage collection proportional to the size of the requested memory block is performed. No GC work is performed when high-priority processes request memory. The missing work is instead performed when no high-priority process is ready to execute.

A minimum GC ratio, GCR_{min}, is again defined as:

$$GCR_{min} = \frac{W_{max}}{F_{min}} \tag{4.22}$$

W_{max} is the maximum amount of GC work necessary to complete a full GC cycle. It includes the work necessary for marking as well as the work required for compacting the heap. F_{min} is the minimum amount of memory available for allocation of new objects during one GC cycle. F_{min} is calculated as:

$$F_{min} = S_B - M_{HP} \tag{4.23}$$

S_B is the size of each of the allocation areas, B1 and B2. M_{HP} is the amount of memory that can be allocated by high-priority processes during a time interval equal to the response time of the GC process.

The garbage collector is scheduled in such a way that the minimum GC ratio is made to be lower than or equal to the current GC ratio, GCR, i.e. $GCR \geq GCR_{\min}$. GCR is defined:

$$GCR = \frac{W}{A} \tag{4.24}$$

W is the amount of work performed so far during the on-going GC cycle and A is the amount of memory allocated during the same period.

When the compaction phase has finished, the garbage collector is suspended until both the current allocation area, B1 or B2, and the area A is filled up, after which a new GC cycle begins. The areas is considered to be filled up when it can no longer be guaranteed that M_{HP} bytes of memory can be allocated without performing GC work. M_{HP} consecutive bytes of memory must remain free in both allocation areas in order to do this, assuming the memory manager does not have any information on maximum object size. If we denote the amount of free memory in A and the current allocation area A_{free} and B_{free} respectively, a new GC cycle is commenced when both of the following condition holds:

$$\begin{aligned} A_{free} &< M_{HP} \\ B_{free} &< M_{HP} \end{aligned} \tag{4.25}$$

The schedulability of the GC work is analysed in the same way as for the copying GC algorithm, see Section 4.8. The value of M_{HP} follows from the analysis.

4.10 Generation-based garbage collection

Using a generation-based GC algorithm is a well-known technique to achieve good average-case performance, as described in Section 3.5. In hard real-time systems, we in general concentrate on minimizing the *worst-case* costs for garbage collection. Techniques, such as generation-based garbage collection, that improve the average-case performance on the expense of the worst-case costs are less useful. For generation-based garbage collection, the worst-case costs increase due to the extra administration required and the potential of GC work in one generation triggering work in another generation (see [Ben90]). Consequently, they have not been considered appropriate for hard real-time systems.

The low interest in generation-based algorithms may be a result of a somewhat simplified view of real-time systems. People tend to divide real-time systems into two distinct classes, *hard* real-time systems and *soft* real-time systems. In a hard real-time system, all processes are assumed to have hard deadlines. The processes of soft real-time systems may also have deadlines, but it is tolerated that they are missed occasionally. As we have observed earlier, most embedded control systems

contain processes with hard deadlines as well as processes with soft deadlines and do thus not really fit well into any of the two classes.

Generation-based algorithms will *not* improve the worst-case performance of a real-time system and do consequently not improve the situation for the hard real-time part of the system. They do, on the other hand, promise improved average-case behaviour, which the soft real-time part of the application will benefit from. It is important, however, that any additional overhead for the hard real-time processes is small enough to not prevent the application from meeting hard deadlines.

Our semi-concurrent scheduling strategy is applicable to most incremental algorithms. Since our scheduling principle incurs delays of any significant length only for the *soft* real-time part of the system, it seems possible to combine incremental generation-based algorithms such as those presented in [LH83, Ben90] with our strategy. The rationale for *not* using generation-based algorithms is thus weakened significantly.

Scheduling the work of an incremental generation-based algorithm according to our scheduling principle brings with it more or less the same types and amounts of overhead as described in Section 4.5. An additional piece of overhead for generation-based algorithms as compared to non-generation-based algorithms is that the write barrier must monitor assignments that create pointers from one generation to another. Tables must be updated to keep track of such pointers. This does, however, *not* lead to more overhead for a garbage collector scheduled according to our strategy than for one using traditional scheduling.

The overall performance of a generation-based garbage collector scheduled according to our principles can be assumed to be more or less comparable with a traditionally scheduled collector.

4.11 Summary

This chapter presented a GC scheduling strategy that permits the use of garbage collection even in hard real-time systems, particularly embedded control systems.

The strategy, which we call semi-concurrent garbage collection, is a combination of concurrent garbage collection with respect to high-priority processes and incremental sequential garbage collection with respect to low-priority processes. The effect of the strategy is that only a small fixed amount of administrative work is performed during execution of critical high-priority processes. This adds fixed overhead to allocation, pointer assignment, and pointer dereferencing, and contributes only with a small amount of time to the worst-case execution time of high-priority processes. The GC work motivated by the allocation activities of the high-priority processes is performed in between the execution of the high-priority processes. The GC work motivated by allocations made by low-priority processes is performed as a part of the allocation operation. This arrangement guarantees that high-priority processes do not run out of free store due to the actions of low-priority

processes. All GC work is interruptible and consists of very small atomic operations, which guarantees that high-priority processes can be invoked without delay.

It was studied how a standard incremental sequential GC algorithm, in our case a variant of Brook's algorithm, can be modified to meet the requirements of semi-concurrent scheduling. It was discussed how alternative algorithms, e.g. incremental mark-sweep and generation-based algorithms, can be used together with the proposed strategy. An advantage of using semi-concurrent garbage collection is that techniques for improving the average-case performance, such as generation-based garbage collection, can be used with virtually no penalty on the worst-case behaviour of critical processes.

The scheduling strategy described in this chapter meets the design requirements for a garbage collector for embedded systems (described in Section 4.2):

- Small and fixed overhead for primitive memory management operations adds minimally to the worst-case execution times of high-priority processes.
- Short latencies for high-priority processes are guaranteed.
- The flexibility in algorithm choice promises good overall effiency.

A priori scheduling analysis is necessary in order to convince ourselves that a particular application program will always meet its deadlines. The chapter showed how a standard scheduling analysis method, in this case rate monotonic analysis, can be modified to take garbage collection into account. Joining the fields of GC research and scheduling analysis research is a novel and important contribution of this thesis. Semi-concurrent garbage collection has a very nice property when it comes to scheduling analysis. We only have to analyse the high-priority processes (which are few in most cases) and the special GC process in order to determine the schedulability of the safety-critical parts of the application. The low-priority processes can never cause the garbage collector to get behind regardless of their memory allocation rate, which would otherwise imply failure to serve the high-priority processes with fresh memory.

Chapter 5

A Garbage Collection Prototype

Having developed a new scheduling strategy for garbage collection in real-time systems, presented in Chapter 4, we now want to verify that the promises on guaranteed short response times for high-priority processes can be fulfilled. A prototype garbage collector is therefore implemented. The requirement on other aspects of the implementation is only that it should be reasonably efficient. The overall efficiency of the collector has thus not been a major concern and many possibilities for improvement exist.

The prototype is integrated with an existing real-time kernel used for automatic control applications. The implementation work is done in cooperation with the Department of Automatic Control, Lund Institute of Technology. This chapter describes the implementation of the prototype while the actual measurements carried out on the prototype are presented in the next chapter.

5.1 Environment

The real-time development environment at the Department of Automatic Control is a flexible and open system [AB91]. It supports several hardware configurations and can be used to control complex systems such as industrial robots. The platform is very suitable for experiments with new implementation techniques for real-time systems due to its openness. The source code for all of the system software is available. This makes it easy to perform experiments in the innermost part of the real-time kernel, which is usually impossible when a commercial system is used.

5.1.1 System architecture

The real-time development environment is based on a host-target architecture. In the general case, one computer host the development tools while another executes

the application program. Several hardware architectures are supported. Unix workstations from Sun Microsystems or IBM PC compatibles can be used as host machines. For the target machine one can chose between a Sun workstation, an IBM PC compatible, or a VME board equipped with a microprocessor from the Motorola 680x0 family. The VME boards are special-purpose control computers without facilities for secondary storage or direct user interaction.

The hardware configuration supported by the GC prototype is a Sun workstation as the development machine and a VME board equipped with a 25 MHz 68040 processor as the target machine, see Figure 5.1[1]. Development tools such as text editors, compilers, and linkers run on the Sun workstation. The workstation is connected to the target VME board via an ethernet connection and a serial RS-232 link. The serial link is used to communicate with a simple ROM-based program loader and a machine code debugger on the VME board. It also provides support for serial I/O while an application is running. Applications are loaded into the VME board computer via the ethernet connection, after which they are given full control of the hardware. The VME hardware includes a number of different analog and digital I/O ports used for interfacing with the hardware that is to be controlled.

5.1.2 Real-time kernel

The real-time kernel supports concurrent priority-based preemptive processes. The system employs a shared-memory model for process communication. Support is included for the most popular synchronization mechanisms, such as semaphores,

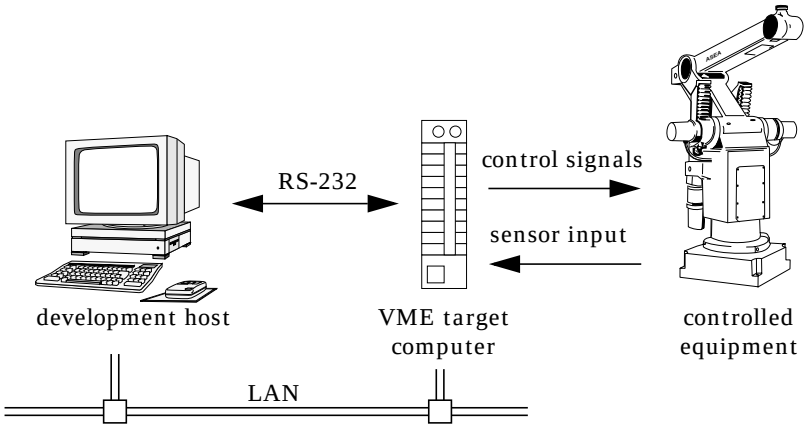


Figure 5.1 Overview of the hardware architecture. Workstations are used to develop applications; compiling, and linking. The linked application is loaded into a dedicated control computer via the network.

1.The picture of the robot in Figure 5.1 is taken from the ASEA/ABB Irb-6 robot manual and digitized by Rolf Braun.

monitors, and message passing. Monitors implement the basic inheritance protocol for priority inheritance in order to avoid problems caused by priority inversion.

The kernel is written in Modula-2, which a Modula-to-C translator converts into C code. The generated C code is then compiled into object code by gcc, the GNU C compiler. The compiled kernel can be used together with application programs written in Modula-2, C, or C++. The real-time kernel object file is linked with the application program into a complete executable program. The resulting program can then be loaded over the network into the target control computer and executed.

5.2 The garbage collector

A garbage collector for Motorola 68040-based VME boards was developed. The garbage collector was added to the existing system with minimal modification of the runtime system and real-time kernel. The compiler, for C or C++, does not offer any special support for garbage collection. Therefore, the programmer is responsible for making sure that pointers and objects on the garbage collected heap are manipulated in the way prescribed by the GC algorithm. This leads to some overhead, both regarding code and performance. Furthermore, the required source code is tedious to write. However, the approach is only intended for experiments. Using the collector on a larger scale would require a more cooperating environment with a programming language designed for use with garbage collection, such as Java or Simula.

5.2.1 The algorithm

The prototype garbage collector is an implementation of the copying algorithm described in Section 4.3. The lazy-evacuation scheme of Section 4.4.2 is implemented in order to minimize the worst-case response times for high-priority processes. The object initialization strategy of Section 4.4.1 is not yet implemented, however. The contents of new objects are initialized to zero in connection with allocation. Object copying can be aborted in order to guarantee small and bounded worst-case latency of high-priority processes, as described in Section 4.4.5. The collector is mainly written in C, but with a few critical pieces coded in assembly language.

5.2.2 The garbage collector coroutine

Coroutines can be viewed as strongly synchronized light-weight processes, where execution is explicitly transferred from one coroutine to another [Mar80]. The execution is transferred to the point where it was suspended last when a coroutine is resumed. Switching from one coroutine to another is typically a cheap operation

compared to switching between two concurrent processes. The cost is usually comparable to that of a procedure call.

GC algorithms can often be elegantly described in the form of coroutines. A full GC cycle for a copying algorithm involves several separate activities performed in sequence; waiting for memory to fill up, performing a flip, scanning root pointers, and scanning evacuated objects. Using coroutines, this work can be implemented by one sequential piece of code. Work can be suspended at arbitrary (but programmer-defined) points in the sequence in order to implement incrementality. When the coroutine is resumed, it continues at the point where it was suspended last. Using coroutines, the GC algorithm can be nicely implemented separately from the scheduling policy.

The mutator and the collector of incremental GC algorithms are often viewed as two separate coroutines. This approach is applicable to our system as well, albeit with a slight modification. In our case, the "mutator" coroutine is composed of all the application program processes plus the special GC process responsible for cleaning up after the high-priority processes (Section 4.4.3). It is important to note that the notion of coroutines is orthogonal to the notion of real-time processes in this case (Figure 5.2). The processes call the same GC coroutine when they have detected that GC work is required. This can be done either by the high-priority GC process when a high-priority process is suspended or in connection with allocation requests made by low-priority processes. The GC coroutine is called repeatedly until enough GC work has been performed. Each time the coroutine is invoked, it performs one increment of GC work.

Some languages support coroutines, such as Simula [SIS87] and Beta [MMN93], but unfortunately, C does not. A C function was therefore used to emu-

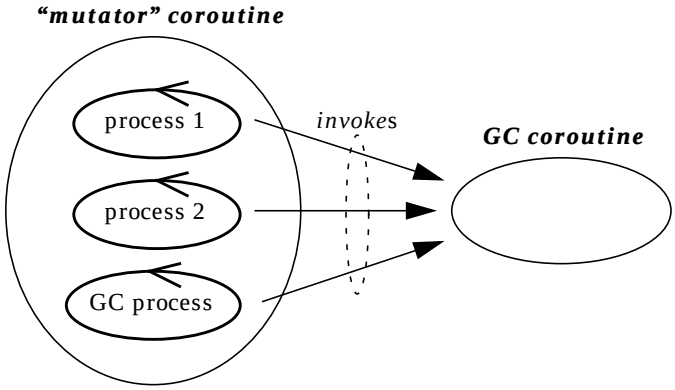


Figure 5.2 The application program processes together with the high-priority GC process together comprise one coroutine. Another coroutine, the GC coroutine is responsible for performing GC work. The notion of real-time processes is orthogonal to the notion of coroutines. Each real-time process executes parts of both coroutines.

late a coroutine. Calling the function corresponds to resuming the coroutine. The first thing performed by the C function is to determine the position where the coroutine was suspended last and jump to that location. A global state variable is used to record where the coroutine was suspended last. The function acts like a finite state machine implemented by a large case-statement. The coroutine relinquish control by returning from the function after having recorded in the global state variable where execution is to be resumed next time the function is called. Locally used variables were declared static in order to preserve their values between invocations.

The C function implementing the GC coroutine does not contain any logic for deciding when to suspend garbage collection. It only performs one small increment of GC work and returns. The logic is instead located at the call-sites of the function. This means that the GC coroutine will be called repeatedly until enough GC work has been performed. Each increment of GC work will thus require one procedure call to be performed. In addition, dispatch code must be executed by the function in order to resume the GC coroutine at the right place. This overhead will be quite significant since the GC work performed at each invocation is very small. We have traded performance for ease of coding. The GC coroutine and its invocation mechanism are obvious targets for future optimization.

5.2.3 Memory organization

The original memory organization of the real-time kernel and development language is used with minimal modification. Apart from sharing statically allocated global data, all processes share a common heap managed by the kernel, also known as the standard C heap or the system heap. Manual memory management (using malloc/free) is used to handle the objects on this heap. Each process is assigned a separate stack, which is used to hold the activation records for the process in question. When a context switch is required, the kernel stores the current value of the stack pointer into the process record of the process to be suspended. Then, the new value of the stack pointer is loaded from the process record of the process to be activated. It is thus possible to change the active stack without copying the contents of the stacks involved. A process stack is allocated dynamically on the manually managed heap each time a new process is created.

A part of the system heap is allocated by the garbage collector at start-up. This memory area is used for the garbage collected heap on which all the garbage collected objects are stored. The application program is free to chose whether a new object should be allocated on the garbage collected part of the heap or on the manually managed heap provided by the original kernel. Figure 5.3 illustrates the memory organization.

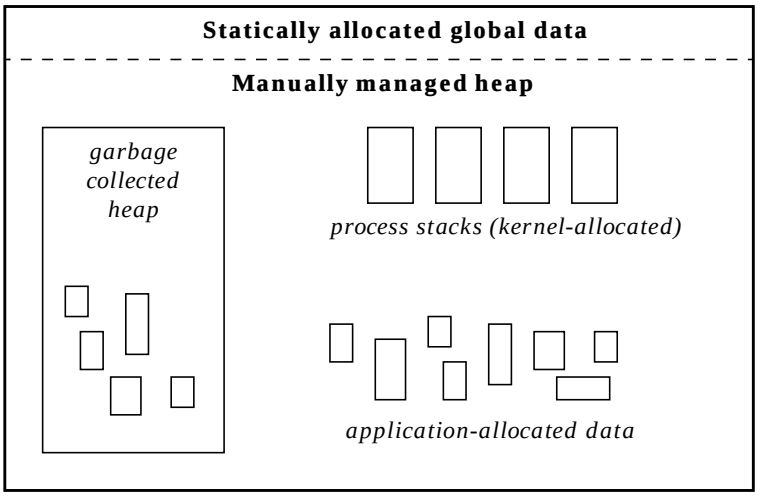


Figure 5.3 Memory organization. The garbage collected heap is allocated at start-up on the standard system heap. The application program can chose to allocate objects on the garbage collected heap or on the manually managed system heap. Process stacks are dynamically allocated on the system heap when new processes are created.

5.2.4 Root pointer data structures

A root pointer is defined as a pointer located somewhere outside the garbage collected heap and containing a pointer to an object on the garbage collected heap. The pointer can be located in the static data area, on one of the process stacks, or in an object allocated directly on the standard C heap. The garbage collector uses two different data structures to keep track of root pointers depending on the location of the root pointer. A set of stacks is used to keep track of global statically allocated roots and roots located on the processor stacks. A linked list is used to track roots allocated on the standard C heap.

As mentioned previously, each process has a separate stack segment used for allocating activation records representing function calls. Roots are introduced on the stack if a function that contains local pointers is called or if any of the parameters to the function is a pointer. The roots are created in a stack-like fashion, which makes it possible to use a stack to keep track of the roots. The address of each new root pointer is pushed onto a stack when created. The addresses are popped from the stack when the corresponding roots are removed. Each process stack has a unique corresponding root pointer stack. The process stack grows from high addresses towards lower addresses. The corresponding root pointer stack is

located in the opposite end of the stack segment reserved for the process, growing towards higher addresses. The root pointer stacks are in turn linked together in a double-linked list, see Figure 5.4. The garbage collector traverses the root pointers by going through the root stacks in the list one by one. The contents of each stack is traversed linearly. Each location on the process stack indicated by the addresses on the root pointer stack is scanned.

A different technique is required to keep track of root pointers located on the standard C heap. For this, a double-linked list is used, which makes it possible to deregister root pointers in random order. Since we did not want to introduce additional dynamically allocated notice objects, it is required that two words of memory is reserved for use by the garbage collector directly following each root pointer. The two words of memory is used to store pointers to the next and previous root pointer in the list (refer also to page 111).

5.2.5 Real-time kernel modifications

We were able to integrate the garbage collector with the existing real-time kernel with very little modifications of the kernel. The kernel was modified on only three points. First, the garbage collector must be informed when a new process is created

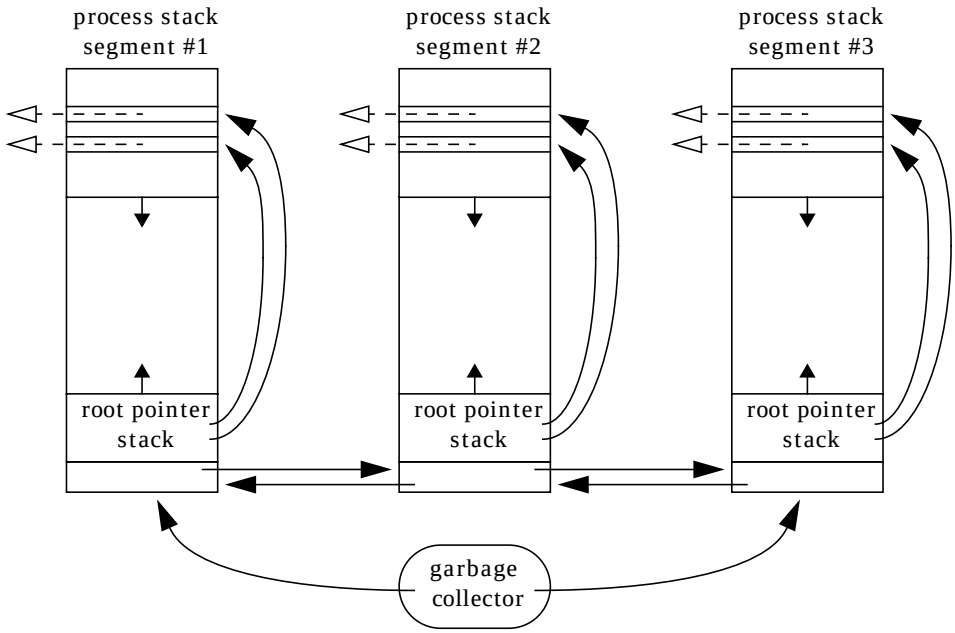


Figure 5.4 Data structure used to keep track of root pointers located on the process stacks and in global memory. Example with three live processes.

in order to create a root pointer stack. Second, the GC process responsible for cleaning up after high-priority processes must be activated whenever a high-priority process is suspended and no other high-priority process is ready to execute. Third, the idle process of the kernel was modified to enable background garbage collection.

Root pointer stacks

As noted above, a root pointer stack is associated with each application process. The root pointer stack must be initialized and linked into a list of such stacks when a process is created. It was therefore necessary to modify the process instantiation routine of the kernel.

The high-priority garbage collection process

As described above, the GC work motivated by the actions of the high-priority processes is performed by a separate process calling the GC coroutine. This process is to be activated when the high-priority processes have allocated memory and GC work is required. One way to accomplish that would be to use to one of the available synchronization primitives supported by the kernel. It was, however, considered unnecessary expensive to signal the GC process using these primitives every time an allocation was made by a high-priority process. Our goal was to minimize the response time of the high-priority processes. Thus, that approach was abandoned. Instead, the process scheduler was modified so that it is aware of the existence of the GC process. When a high-priority process is suspended, the kernel makes the GC process ready to run if not already so, i.e. it moves the process to the “ready”-queue of the scheduler. The scheduler will then assign the processor to the GC process as soon as no high-priority process is eligible for execution.

The advantage of the chosen strategy over explicit signalling is that it does not add execution time to the high-priority processes. On the other hand, the GC process may be triggered unnecessarily if none of the high-priority processes requested an allocation, but that will not add to the response times of the high-priority processes. It can, however, be avoided if the allocation operation sets a dirty bit, which can then be checked by the scheduler. The GC process would then be activated only if the dirty bit is set. Setting a dirty bit in connection with allocations would not significantly add to the response time of a high-priority process. This optimization is, however, not yet implemented.

Garbage collection and idle time

A natural optimization of the garbage collector is to use idle processor time to perform GC work, which means that the garbage collector will be allowed to work

ahead of its schedule. When an allocation request later arrives, the required GC work might already have been performed, resulting in better average-case performance primarily for low-priority processes. The idle process of the kernel was modified to call the garbage collector coroutine repetitively until no GC work remains to be done. This feature was, however, not enabled during the evaluation of the garbage collector since we wanted to provoke as bad situations as possible.

5.2.6 Estimating garbage collection work

The amount of GC work that has been performed can be measured in several ways, as discussed in Section 4.7. The current implementation uses a combination of three parameters to estimate the GC work: the number of scanned root pointers, $roots_{scan}$, the accumulated size of the scanned heap objects, $objects_{scan}$, and the accumulated size of the evacuated objects, $objects_{evac}$.

The current amount of performed GC work is then calculated using the formula:

$$W = \alpha \cdot roots_{scan} + \beta \cdot objects_{scan} + objects_{evac} \quad (5.1)$$

The two coefficients α and β are used to compensate for the fact that scanning a root pointer or a heap object takes a different amount of time to perform than evacuating an object. Correctly chosen, α and β make W reasonably proportional to the actual time spent on GC work. Empirical studies have given that 7 and 0.5 seems to be reasonable values for α and β respectively for our implementation and hardware.

Denote the corresponding upper bounds on the variables used above max($roots_{scan}$), max ($objects_{scan}$), and max($objects_{evac}$) respectively. The maximum number of roots is explicitly provided by the programmer. The latter two bounds are equal to the maximum amount of simultaneously live memory, which is also programmer-specified. Then, the maximum amount of required GC work is calculated similarly to the amount of currently performed work:

$$W_{max} = \alpha \cdot max(roots_{scan}) + \beta \cdot max(objects_{scan}) + max(objects_{evac}) \quad (5.2)$$

The values of W and W_{max} derived from (5.1) and (5.2) can now be applied to the relation in (4.5) on page 59 in order to determine how much GC work must be performed.

5.3 Application program interface

The application program interface, or API, to the memory management module consists of a C interface with some extensions for C++ constructs. Lacking a cooperating compiler, the programmer must follow a number of coding conventions to guarantee the consistency of the heap at all times. Here, we briefly describe the API of the memory manager and the conventions the application programmer has to adhere to.

5.3.1 Initialization

Before any object can be allocated on the garbage collected heap, the memory manager must be initialized. This is done by calling the routine *gc_init* once at start-up:

```
gc_init(int space_size,int live_size,int max_roots,int hp_alloc,
        int hp_prio_limit,int gc_during_idle);
```

The *space_size* and *live_size* arguments specify the size of the GC heap and the maximum amount of simultaneously live memory respectively. The maximum number of root pointers is given in *max_roots*, information that is used by the garbage collector to schedule its work. *hp_alloc* specifies the amount of memory that is to be reserved for use by high-priority processes, M_{HP}, see Section 4.8.3. Processes with a priority higher than the priority given in *hp_prio_limit* are considered to be high-priority processes. The high-priority GC process is assigned the priority in *hp_prio_limit*. Low-priority application processes must have a lower priority. The last argument, *gc_during_idle*, is a flag determining whether the garbage collector should be invoked when the systems idle process is running. This is an option that improves the average-case behaviour of the collector, since it allows the collector to do work before it is actually required by the scheduling rules. It does not improve the worst-case behaviour, however. Garbage collection during idle processor time should normally be enabled, but we included the option to disable it for experimental reasons. We ran the collector with this feature disabled in all the experiments described in the next chapter in order to stress the collector as much as possible.

5.3.2 Declaring objects

All objects (C structs) located on the GC heap must contain a header with a fixed layout. The header must contain four word-sized fields. The following example illustrates what a garbage collected C data structure looks like.

```
typedef struct example_object {
   /* Header fields */
   struct example_object *fp;          /* Forwarding pointer */
   int                   object_size;
   int                   *gc_info;     /* Pointer to object  */
                                       /* layout description */
   void                  *gc_flags;    /* GC status and lazy */
                                       /* evacuation pointer */
   /* Application-specific fields */
   ...
} example_object;
```

The *fp* field stands for “forwarding pointer”. All access to heap objects must be done via this pointer. The total size of the object, including the header fields, can be found in *object_size*. The *gc_info* field is a pointer to information about where pointers are located in the object. We describe this further below. The last field, *gc_flags*, is used by the garbage collector during a GC cycle to determine whether the object is evacuated, marked for lazy evacuation, or yet unmarked. If the object is marked for lazy evacuation, *gc_flags* contains a pointer to the memory area in tospace reserved for the object.

Object layout information

When the garbage collector scans an object for pointers to other objects, it must be able to correctly identify the pointer locations within the object. To facilitate this, the *gc_info* field in the object header contains a pointer to a data structure with information about the layout of the object. All objects with the same pointer layout can share the same layout information. The layout information is parsed by the garbage collector as the object is scanned.

The layout information consists of a sequence of 32-bit integer pairs terminated by a single integer having the value -1. Each integer pair consists of an *offset* and a *count*. The offset indicates how many 32-bit words an imagined scan pointer, initially positioned at the start of the object, should be advanced in order to be positioned at to the next application program defined pointer field. The count states how many consecutive pointer fields follows, and how much the scan pointer is to be advanced before the next integer pair is parsed. Figure 5.5 presents an example object declaration and the corresponding layout information.

A count of -2 indicates that a variable number of pointer fields follows. The actual number of pointers is found within the object at the position indicated by the scan pointer. The pointer fields follow immediately thereafter. This makes it possible to use the same layout information for objects with a varying number of pointer fields, such as arrays.

The layout information about an object must currently be constructed manually by the programmer. In a cooperating environment, in which the compiler is aware

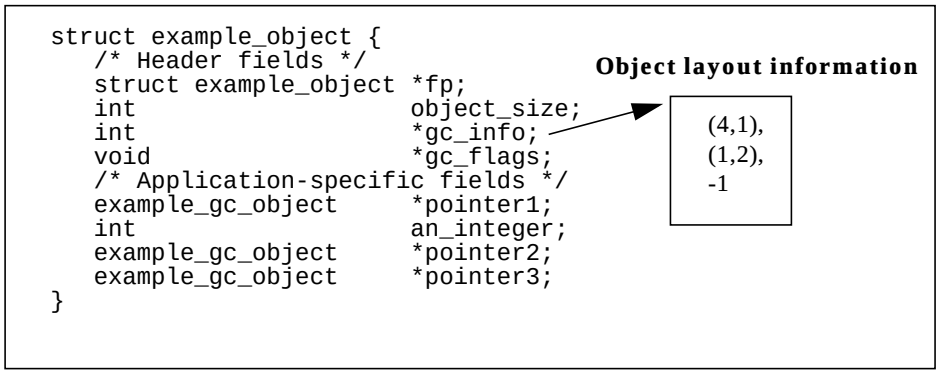


Figure 5.5 An example object declaration and the corresponding object layout information. The first integer pair (4,1) indicates that four words should be skipped in order to find the first pointer and that only one pointer follows. The next integer pair (1,2) indicates that two pointers follows after one non-pointer word.

of the garbage collector and its requirements, this information would be generated automatically.

5.3.3 Pointer access

Any transaction that temporarily creates pointers into the GC heap that the garbage collector is not aware of must be protected. Otherwise, the garbage collector might move the referenced object without updating the temporary pointer. One such transaction is pointer access, which might put temporary pointers in processor registers. Protection is achieved by disabling the processor interrupts during the access, thereby preventing the garbage collector to get control. Two C macros exist for disabling and enabling interrupts, *gc_lock* and *gc_unlock* respectively.

Pointer dereferencing must always be performed via the indirection pointer *fp* in the object header, as mentioned in Section 5.3.2. This implements the read barrier of the GC algorithm. The code for accessing a field aField in an object reference by a pointer *aPtr* might thus look like:

```
gc_lock;
a = aPtr->fp->aField;
gc_unlock;
```

Implementation note: The *gc_lock* and *gc_unlock* macros expand into inline assembler code. In the current implementation, *gc_lock* generates two machine instructions and *gc_unlock* one machine instruction.

5.3.4 Pointer assignment

Pointer assignments must be guarded by the write barrier of the copying GC algorithm. A macro named *gc_set* is used to implement pointer assignments. All assignments to pointers must be performed using this macro. The macro has the syntax:

```
gc_set(lvalue,pointer expression);
```

The pointer resulting from the evaluation of *pointer expression* is assigned to the pointer given as *lvalue*.

Implementation note: The gc_set macro implements the lazy-evacuation scheme described in Section 4.4.2. In the worst case, 21 machine instructions must be executed in the current implementation including code for disabling/enabling interrupts. The code generated for evaluating the arguments is not included.

5.3.5 Allocation

New objects are created using a call to *gc_new*:

```
gc_new(int *gc_info,int size,void *target);
```

A pointer to the layout information for the new object is passed in *gc_info*. The size of the requested object is given in *size*. When the new object has been created, a pointer to it is stored in the pointer location denoted by *target*.

5.3.6 Root pointers

Root pointers must be registered in the data structures of the garbage collector before they are used (refer to Section 5.2.4). In our C implementation, root pointers come in three variants: Roots located on stacks (local variables and function parameters), roots in static global memory (global variables), and roots dynamically allocated on the standard C heap.

Local pointer variables

Roots are allocated on the process stacks as function calls creates activation records on the stacks. If a procedure contains a declaration of a local pointer variable, it will be instantiated on the stack as a part of the activation record. The programmer is responsible for informing the garbage collector about these new root pointers before using them for the first time. This is done by using the

gc_pushroot macro. The macro must be called once for each local pointer variable and takes the local variable as argument. The addresses of the local variables are then pushed onto a root pointer stack. A separate root pointer stack is associated with each process. Another feature of the macro is that the pointer is initialized to null.

Before returning from the function, the programmer must remove the temporary root pointers from the root pointer stack. This is done by using the *gc_poproots* macro, which removes the number of roots from the stack given as argument. A code of a sample procedure containing declarations of local pointer variables might look like:

```
void example_routine(void) {
   example_object *local_root1,*local_root2;

   gc_pushroot(local_root1);
   gc_pushroot(local_root2);
   ...
   gc_poproots(2);
}
```

Global pointer variables

Root pointers located in static global memory are live from the start of the application and until it is terminated. The same technique is used to inform the garbage collector about global roots as locally declared roots. The main process is responsible for calling *gc_pushroot* for each global root. This is done after having initialized the garbage collector with a *gc_init* call. References to global roots will thus be available on the root pointer stack of the main process throughout the execution of the application.

Passing pointers as parameters

When a pointer to an object on the GC heap is passed as a parameter to a function, the parameter forms a new root pointer on the process stack. As such, it must be registered in the garbage collector. The problem facing the application programmer is that this can not be easily done before calling the function. The parameter locations are only directly accessible within the function itself. In order to avoid a context-switch while the parameters are evaluated and the function is called, potentially causing the garbage collector to move the involved objects, the interrupts must be disabled until the parameters can be registered. Complex expressions involving functions calls should be avoided as arguments to functions in order to avoid long blocking times. The parameters are registered one by one by the function itself, after which the interrupts can be reenabled. A sample function taking a pointer as a parameter could look like:

```
void sample_function(Object *ptr) {
   gc_pushparam(ptr);
   gc_unlock;
   ...
   gc_poproots(1);
}
```

The code calling the function might look like:

```
gc_lock;
sample_function(pointer expression);
```

The *gc_pushparam* macro is similar to *gc_pushroot*, described earlier, but it does not clear the contents of the root pointer.

Returning a pointer as the result of a function cause similar problems as pointer parameters do, and the solution is similar. The processor interrupts are disabled before the function is terminated and reenabled when the return value have been safely stowed away in a pointer variable known to the garbage collector.

Dynamically allocated roots

As described in Section 5.2.4, root pointers located on the standard C heap must be handled differently than other types of root pointers. Dynamically allocated root pointers must be declared as a C struct with the following layout:

```
struct type_name {
   object_type      *ptr;
   struct type_name *prev;
   struct type_name *next;
};
```

When dereferencing the pointer, the *ptr* field should be used. The other fields are used by the garbage collector to keep track of the root pointer. The application must register the root before using it for the first time. This is done by calling *gc_add_heaproot*. The root is deregistered by calling *gc_remove_heaproot*.

In the following example, a root pointer is declared, dynamically allocated, used, and finally deallocated:

```
struct PtrType {
   Object         *ptr;
   struct PtrType *prev;
   struct PtrType *next;
};

struct PtrType *aPtr;
...
aPtr = malloc(sizeof(PtrType));
gc_add_heaproot(*aPtr);
...
gc_set(aPtr->ptr,...);
```

```
...
gc_remove_heaproot(*aPtr);
free(aPtr);
```

Implementation note: The garbage collector implementation maintains a double-linked list of all dynamically allocated root pointers. The *prev* and *next* fields of the C struct are used for forward and backwards links. The overhead for the scheme is quite high. Three words of memory are required for every root pointer instead of only one. Furthermore, the application must call routines to insert or remove root pointers from the linked list each time a root pointer is allocated or deallocated. This approach was chosen because it did not require any modifications of the existing compiler or runtime system and is only intended for experimental use. The problem disappears when the garbage collector is tightly integrated with the programming language and runtime system.

5.3.7 Garbage collecting C++ objects

Even though the garbage collector was written in C and have a C API, it can still be used to garbage collect C++ objects. Some restrictions do, however, apply.

Multiple inheritance

The garbage collector assumes that every pointer to an object on the GC heap points to the beginning of the object header. This precludes the use of multiple inheritance in C++, since the implementation of multiple-inheritance creates pointers that points to different parts of the object depending on the static qualification of the pointer [Kro85, Str94].

Declaration and instantiation

All garbage collected C++ objects must contain a four-word header just as C structs must (Section 5.3.2). The contents of the header must be the same as for C structs. If the class contains virtual member functions, the C++ compiler will insert a pointer to a virtual function table at the start of the object [Str94]. This is not what we want since the fields of the GC header would end up at the wrong offsets within the objects. A solution to this problem is to declare an abstract class only containing the GC header fields and use it as a super class to a class containing the virtual member functions. This results in an object layout in which the GC header fields precede the pointer to the virtual function table.

Classes are instantiated using the *gc_new* function. The constructor, if it exists, of the class is consequently **not** called. If an initialization function is required, it

must be called manually. Similarly, when the garbage collector reclaims the memory occupied by the object, the destructor, if present, is not called either.

The 'this' pointer

When a C++ member function is called, an implicit parameter is passed to the function. The parameter is called the 'this' pointer and refers to the object receiving the message implemented by the method. If the receiver of the message is an object on the GC heap, the 'this' pointer must be registered by the garbage collector, just as any other pointer parameter. A special macro, *gc_pushthisroot*, is used to do this. It is used similar to *gc_pushparam* but it takes no argument; it is implicit.

Implementation note: The location, i.e. the memory address, of the 'this' pointer cannot be directly accessed from C++ code. Therefore, the *gc_pushthisroot* macro had to be written in assembly language. It makes an implementation-dependent assumption about where the 'this' pointer is located with respect to the activation record of the currently executing function.

Member access within member functions

As pointed out in Section 5.3.3, all pointer dereferencing must be performed via the forwarding pointer of the target object and with the processor interrupts disabled. This naturally applies to C++ objects as well. When a member variable is accessed from outside the object, it is very obvious that an pointer is dereferenced. However, in C++, a member access from within a member function does not require an explicit pointer dereferencing. The 'this' pointer is implicitly dereferenced in connection with the member access. In our implementation, every access to a member entity must be done via the forwarding pointer and with interrupts disabled, even from within member functions. This is illustrated in the following example:

```
class sample_class {
   public:
      sample_class *fp;
      int          object_size;
      int          *gc_info;
      void         *gc_flags;
      int          a;

      int          get_a();
};

int sample_class::get_a() {
   int tmp;

   gc_pushthisroot; // Register the “this” pointer as a root
   gc_unlock;       // Permit preemption here
```

```
        gc_lock;
        tmp = fp->a; // Fetch the value of a via the forwarding pointer
        gc_unlock;   // Permit preemption here

        gc_poproots(1); // Deregister the “this” pointer
        return tmp;
    }
```

5.4 Discussion

The existing prototype garbage collector is only very loosely coupled with the rest of the development environment. The motivation for this was to minimize the implementation work necessary to perform the measurements, but it has some serious drawbacks. The loose coupling to the compiler often leads to suboptimal code being generated. Very little optimization has been made in order to gain speed or to minimize the space requirements. The programmer must follow a detailed set of coding restrictions in order to manipulate the garbage collected objects in a safe way. Together, this results in a system that is not suitable for use in production-quality applications. However, it is good enough to be used to verify that the scheduling strategy presented in this thesis is indeed practically useful.

A language system, compiler and runtime system, with built-in support for garbage collection is necessary for applications to really benefit from garbage collection. C and C++ are not safe and are not suitable languages for such applications. A safe language such as Java, Simula, Beta, or Eiffel would be an adequate choice. For these languages, the compiler can ensure that every restriction on how pointers are used are met and can automatically generate the necessary code. The tedious and error-prone programmer overhead for coding according to the restrictions would be eliminated. The system we used for experiments was, however, already implemented in C/C++ and in this situation the chosen approach was judged to be workable.

One of the key motivations for using garbage collection in hard real-time systems is to increase robustness. The increase in robustness is achieved by relieving the programmer of the responsibility of manually managing the memory. The problems of dangling pointers, memory leaks, and memory fragmentation are eliminated.

Another advantage of a more tightly integrated system than the current implementation is that more efficient code can be generated by the compiler. An example of how a cooperative compiler could yield better code is object access. Currently, context switches must be prevented while a pointer is dereferenced and a field within an object is accessed (Section 5.3.3). The implementation requires the programmer to call the *gc_lock* macro, disabling interrupts, before executing a statement that dereferences a pointer. The *gc_unlock* macro is used to reenable the interrupts afterwards. The following code illustrates how a pointer access must be protected:

```
gc_lock;
a = sqrt(b)/sin(ptr->fp->c+d);
gc_unlock;
```

Apart from making the explicit *gc_lock*/*gc_unlock* calls unnecessary, a compiler that is integrated with the garbage collector could improve the code generated for the assignment above in several ways. We observe that the interrupts are disabled during the execution of the entire assignment statement. This is not really necessary. It is only the part of the statement that dereferences a pointer that has to be protected, i.e. *ptr->fp->c*. A compiler that automatically inserts code to disable/reenable interrupts would realize this and generate code accordingly. Another alternative would be to reserve one or two processor registers for storing the intermediate pointers needed dereferencing the pointer. A garbage collector that is aware of this would then consider those registers to be roots and scan them as any other root pointer. It would not be necessary to disable the interrupts at all if such an approach was used. Other coding conventions, e.g. pointer assignment, could be optimized in similar ways.

5.5 Summary

A prototype implementation of a garbage collector scheduled according to the principles described in this thesis was made. The purpose was to investigate whether our techniques are useful in actual control applications. An existing real-time kernel was modified, adding a real-time garbage collector. The garbage collector was coded in C with some critical loops in assembler. Even so, the aim of the implementation was not to achieve maximum performance, but simply to demonstrate the feasibility of our ideas.

The algorithm used is a variant of Brook's incremental copying algorithm, although other algorithms can be used instead. This algorithm was chosen for its simplicity, not for efficiency reasons.

The garbage collector implementation is library-based, which means that it is only loosely coupled with the language and compiler used to develop application programs. The programmer is therefore required to follow a detailed set of coding conventions in order to guarantee correct behaviour by the garbage collector. An overview of the programming interface and the associated coding conventions was given.

Chapter 6

Experimental Results

In this chapter we report on the experiments and performance measurements carried out with the garbage collector prototype described in the previous chapter.

6.1 Introduction

The purpose of the experiments described in this chapter is to verify that a garbage collector can be implemented according to the ideas presented earlier in this thesis and that such an implementation meets the requirements of hard real-time systems. Important properties that we want to study include:

- The cost of primitive memory management operations, such as pointer assignment and memory allocation.
- The worst-case latency for invoking a high-priority process.
- The amount of processor time required for garbage collection.

We are interested in determining the *worst-case* costs for various operations. Unfortunately, this is something that cannot be directly measured. It does not matter how long we monitor the execution of an executing control application, we can still not be absolutely convinced that the worst case has occurred. The best we can hope to achieve by measuring the performance of an executing program is to get an approximation of the worst-case costs with a reasonable amount of reliability. When we in this chapter refer to the *worst-case cost* of something, we are actually meaning the worst *observed* cost. This is an important distinction to make and it should be kept in mind when reading the chapter.

6.2 Experimental setup

The target computer used in the experiments is a VME-based control computer equipped with a 25 MHz Motorola 68040 microprocessor. The VME computer has access to a number of digital and analog I/O ports which are used to interface with the external equipment that is to be controlled. An example of such external equipment is an Irb-2000 industrial robot from ABB, Asea Brown Boveri. Such a robot is used in one of the experiments described in this chapter. A more detailed description of the hardware can be found in [Nil96].

In order to measure shorter time intervals (from microseconds down to nanoseconds) than any software-based technique can measure, a set of sixteen digital outputs was used. The real-time kernel and the experimental garbage collector were augmented with code producing signals on the digital outputs during interesting events. For example, the cost of pointer assignments could be measured by preceding every pointer assignment with a machine instruction producing a high signal on one of the outputs. The signal was lowered by another machine instruction following the pointer assignment.

New functionality that allows the execution of application processes to be traced was permanently added to the real-time kernel [Ive98]. Furthermore, clock interrupts, process scheduling work, and GC work can be monitored. Digital output signals are used to indicate which process is currently executing. Each application process we want to monitor is assigned a unique output which goes high as long as the corresponding process is executing.

A 16-channel logic analyser was used to study the generated output patterns. The logic analyser continuously samples the state of each of its sixteen input channels and stores the samples in memory. The result of the sampling is shown on the display of the logic analyser, see Figure 6.1, when a trigger condition is met. The display shows how the different signals varied during a specifiable time interval before and after the trigger condition was met. The duration of interesting events visible on the display can be measured with a high degree of accuracy. The trigger condition can be varied in a flexible way. A simple condition is a channel going high, but complex combinations of signals can also be specified as well as an external trigger signal.

Measuring the worst-case and average-case duration for a signal is somewhat tricky without extra equipment. The logic analyser has two modes that could be useful when measuring worst-case durations, but it lacks functionality for measuring average-case durations. The first method for measuring worst-case times involves configuring the logic analyser to trigger only when signals longer than a minimum duration are encountered. The other method involves preventing the analyser to clear the display each time it is triggered, overlaying all encountered signals. The time from the trigger point to the end of the longest invocation can then be measured. This assumes that every invocation starts at the same position

on the display, which is difficult to achieve in practice when the interval between two invocations is short.

To circumvent the limitations of the logic analyser, a second VME-based control computer was used for determining the worst-case and average-case duration of various events. The output signal we wanted to study was not only connected to the logic analyser, but also to a digital input on the second control computer, see Figure 6.2. A tightly coded program written in assembler sampled the input signal and recorded information about the longest encountered signal and the average length of the signal. Whenever an input pulse was encountered that had a longer duration than any previously observed pulse, a short output pulse was generated on a digital output that was connected to the logic analyser, which was in turn triggered. The length of the longest observed signal could then be measured using the logic analyser.

A drawback with using an auxiliary computer to detect the worst case is the lower precision in the measurement compared to if the logic analyser could be used directly. The program sampling the input signal at the auxiliary computer has a

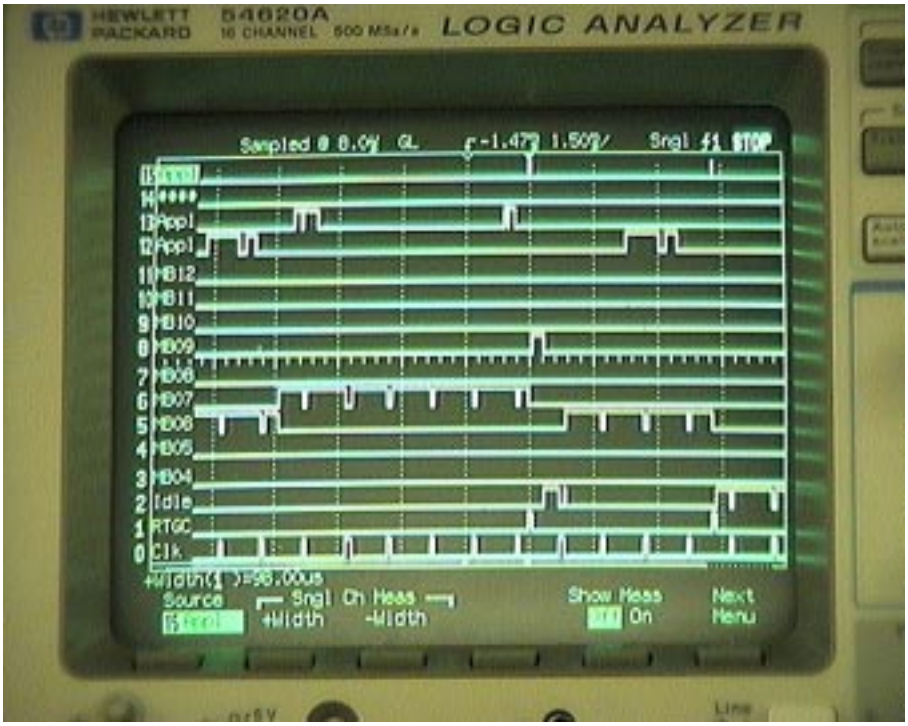


Figure 6.1 Logic analyser displaying a snapshot of running processes. The execution of the prototype real-time garbage collector is shown on the second line from the bottom.

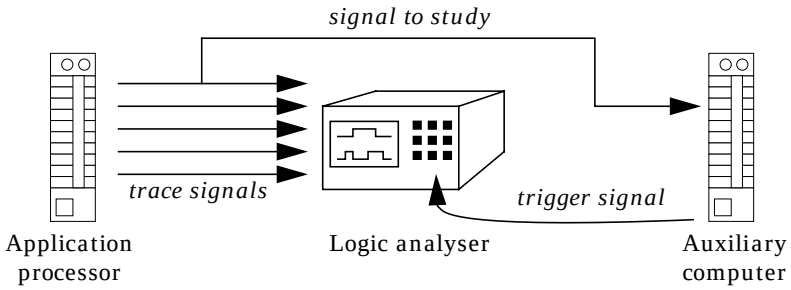


Figure 6.2 Measuring worst-case and average-case duration of events. A logic analyser and an auxiliary control computer are combined in order to produce the desired information.

sampling period of about 1.3 µs. Therefore, the potential error in every measurement is ±1.3 µs.

When the program at the auxiliary computer terminates, it reports the worst-case and average-case signal duration expressed in samples. The worst-case duration can be measured on the logic analyser, but the average-case duration must be calculated. It is calculated by multiplying the number of samples with the sample period. The sample period can in turn be calculated by dividing the measured worst-case duration with the number of samples reported by the auxiliary computer for the worst case.

The processor utilization is also determined using the auxiliary computer. Whenever the idle process on the application computer is running, one of the digital outputs is set high. The signal is continuously monitored by a program executing at the auxiliary computer. The amount of idle time can be determined from the relation between the number of times a high signal was detected and the total number of samples.

6.3 Overview of experimental applications

Five applications were used to test the performance of the garbage collector prototype. Three of them were very small simple applications with the only purpose to test the garbage collector and to see how its performance varies when different parameters are changed. The two remaining programs were existing control applications that were partially converted to using automatic memory management.

SingleHP

The simplest of the test programs will be denoted *SingleHP*. SingleHP is used to study how the costs of individual memory management operations are affected

when different parameters are varied, for example how the cost for allocation requests relate to the object size. It consists of a single high-priority process, invoked at a rate of 200 Hz. The user can specify the size of the garbage collected heap, the maximum amount of simultaneously live memory, and the maximum and minimum object size can be specified when the program is started. An object with a size randomly chosen within the user-specified range is allocated each time the high-priority process is invoked. The allocated objects are inserted into a data structure. The process keeps track of the total amount of currently allocated memory and if it threatens to exceed the maximum amount of simultaneously live memory, objects are removed from the data structure before the new object are allocated. The actual amount of live memory will thus always be very close to the maximum specified amount.

Single LP

This application is identical to SingleHP, except for all processes having a low priority from a garbage collection point of view. The SingleLP application was used to study memory allocation costs for low-priority processes.

GCTest

The *GCTest* program is a program that tests that the system can handle several high-priority processes that allocates and manipulates garbage collected objects. In addition to the high-priority processes, several low-priority processes allocate memory as well. Both high-priority and low-priority processes allocate message objects and send them to a server process. The server process responds to the messages by allocating new objects and inserting them into a global data structure. Objects are removed from the global data structure by a special process that is invoked periodically and which ensures that the amount of currently live memory does not get too high.

Pålsjö - Polynomial

Pålsjö is a system for rapid development of experimental embedded control system prototypes [Eke97]. It is designed to allow on-line configuration and reconfiguration of running control systems. The functionality of periodic control threads is described by block diagrams. Each block has a number of input and output signals. The output signals of a block are connected to the input signals of other blocks. The blocks contain states and new output signals are calculated periodically as the internal states change based on the new values of the input signals and the previous states of the blocks. A special language, the Pålsjö Command Language (PCL), is used to connect the blocks to each other during runtime. The blocks themselves

are usually implemented in PAL, Pålsjö Algorithmic Language, but can also be implemented in traditional programming languages such as C, C++, or Modula-2. Blocks implemented in PAL are translated to C++ by a special compiler.

The Pålsjö system, and one of the standard PAL libraries were partially converted to use automatic memory management. Then, an existing control application that used the converted library was used to test the performance of the garbage collector prototype. The library chosen for conversion implements representation of polynomials and mathematical operations on these. The modified library represents polynomials using dynamically allocated objects managed by the garbage collector. The control application implements an adaptive servo motor regulator. The control law is described by polynomials which are modified in real time. The application relies heavily on polynomial calculations, both to adaptively update the control law and for the process control itself. The application is described in more detail in [Eke97].

Pålsjö - Robot

This program is used to show that it is possible to perform a hard real-time control task in a system with garbage collection. An existing control application based on Pålsjö was modified slightly to rely on automatic memory management for some of its data structures.

An industrial robot, an IRB-2000 from Asea Brown Boveri, was used to control an inverted pendulum, see Figure 6.3. The task of the robot is to pick up the pendulum housing from a table, swing the rod of the pendulum to an upright position, and then balance it in that position. Failure to meet the tight deadlines results in jerky control behaviour or even causes the robot to loose control of the pendulum altogether. It is therefore critical that the garbage collector does not disturb the control processes of the application. The inverted pendulum experiment is described further in [Eke97].

Two high-priority processes are responsible for continuously measuring the current angle of the very unstable pendulum and to calculate new control signals for the joints of the robot. The processes execute periodically with a frequency of 100 Hz and 40 Hz respectively. The processes also generate plot data each time they execute. The plot data is stored in dynamically allocated objects. Low-priority processes handle operator communication and are also responsible for transmitting the plot data generated by the high-priority control processes to a separate computer hosting a graphical operator user interface.

6.4 Measurements of garbage collection costs

Here we study what impact the prototype memory manager has on the execution of the application program. As already mentioned, the costs of individual memory

management operations add to the response times of the application processes and should therefore be kept small, especially for high-priority processes. Another kind of interference is an increased process latency due to locking.

6.4.1 Pointer assignment

The write-barrier used in the prototype relies on lazy evacuation of objects (see Section 4.4.2). This makes the number of machine instructions that have to be executed in the worst case small and bounded. More than 21 instructions must never be executed for a normal assignment. The exact number of required instructions varies somewhat depending on the complexity of the source and target expressions, but the worst-case GC overhead is constant.

The cost of performing a pointer assignment was measured for each of the five test applications and the results are summarized in Table 6.1. Object size, heap size, and the amount of simultaneously live memory were varied when running the

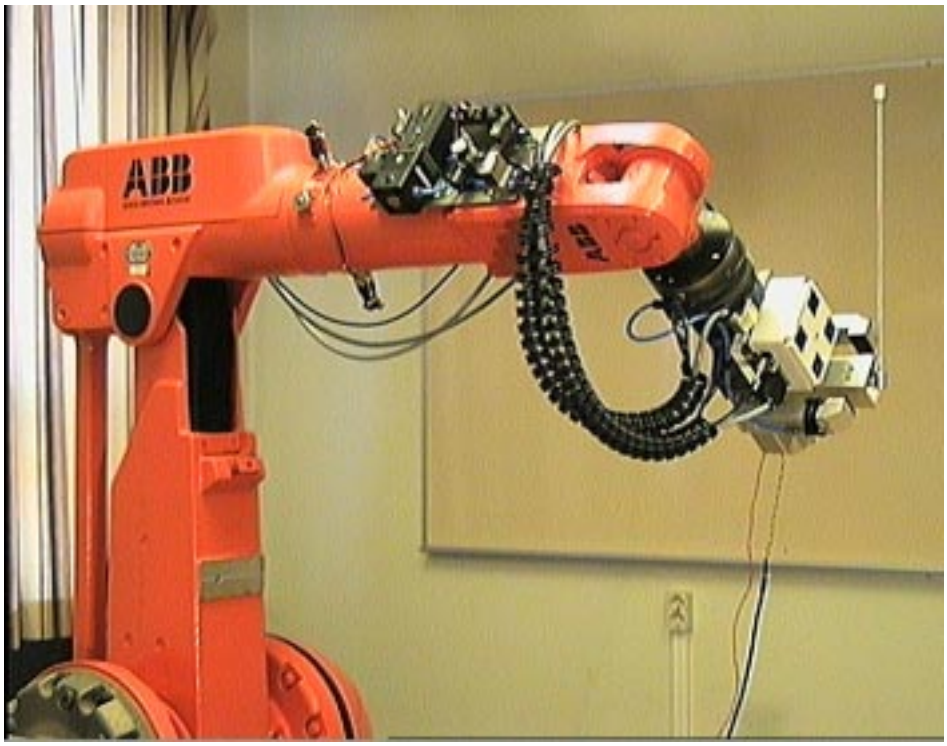


Figure 6.3 IRB-2000 industrial robot balancing an inverted pendulum. The rod of the pendulum (to the far right) can turn freely around the shaft of a potentiometer mounted in the square housing held by the robot.

first test application, SingleHP, but the cost remained unchanged. The cost for pointer assignment is the same for both high-priority and low-priority processes since the write barrier is identical.

SingleHP	*SingleLP*	*GCTest*	*Pålsjö-Polynomial*	*Pålsjö-Robot*
8 µs (3 µs)	8 µs (3 µs)	9 µs (3 µs)	7 µs (3 µs)	7 µs (3 µs)

Table 6.1 The cost of pointer assignments for the five test application programs. The worst-case cost is shown with the average case cost within parenthesis.

6.4.2 Memory allocation

The cost of memory allocation differs depending on the priority of the process requesting memory. No GC work is performed in connection with allocation in high-priority processes, whereas allocation requests made by low-priority processes trigger immediate GC work.

Memory allocation in high-priority processes

The SingleHP program was used to study the cost of allocating an object in a high-priority process. The allocation operation was augmented with measurement code as described in Section 6.2. Three parameters were varied, one at the time. The parameters were heap size, the ratio of live memory, and the object size.

The heap size were varied in four steps from 10000 bytes up to 500000 bytes, i.e. the semispace size varied from 5000 to 250000 bytes. The amount of simultaneously live memory was in each case 20% of the total heap size. A mix of objects ranging from 20 bytes to 400 bytes in size was allocated. The worst-case costs of allocating an object varied between 46 µs and 47 µs. The difference is smaller than the 1.3 µs uncertainty in the measurements. The results thus indicate that the allocation cost is independent of the heap size, which is to be expected.

The ratio of live memory was varied from 5% to 30% with a fixed heap size of 100000 bytes. Again, a range of objects with sizes between 20 bytes and 400 bytes were allocated. All measurements showed that the worst-case cost of allocating an object was 46 µs. The cost is thus independent of the ratio of live memory as well, just as we would expect.

Last, the size of the objects being allocated was varied. We used a heap size of 100000 bytes and a ratio of live objects of 20%. All objects were of equal size, but the size was varied for each test run. We varied the object size from 50 bytes to 5000 bytes. The results are presented in Table 6.2, and Figure 6.4.

According to theory, the amount of work required to allocate an object (in the current implementation) consists of a fixed number of instructions to reserve an

50 bytes	*200 bytes*	*500 bytes*	*1000 bytes*	*5000 bytes*
32 µs	36 µs	50 µs	76 µs	273 µs

Table 6.2 The worst-case cost of allocating objects of varying sizes for a high-priority process when the contents of the new objects are initialized at allocation time. If the responsibility of object initialization was transferred to the collector, as described in Section 4.4.1, the allocation time would be approximately 26 µs independent of object size.

area of memory and to initialize the contents of the object header plus a varying number of instructions to initialize the contents of the new object. No instructions are executed to perform GC work. The measurements confirm the theory and show that the cost of memory allocation is linear to the size of the requested object. Extrapolation gives that the cost function intercepts the y-axis at approximately 26 µs, which consequently is the time required to reserve an area of memory and initialize the object header. This is thus the time an allocation request would require if the collector initializes the contents of the new objects as a part of its normal GC work, as was described in Section 4.4.1.

The first measurement, allocation of 50-byte objects, displays an anomaly that somewhat contradicts the linearity of the cost function. Looking at Figure 6.4, it seems that the function is non-linear at low object sizes. To explain this, it is nec-

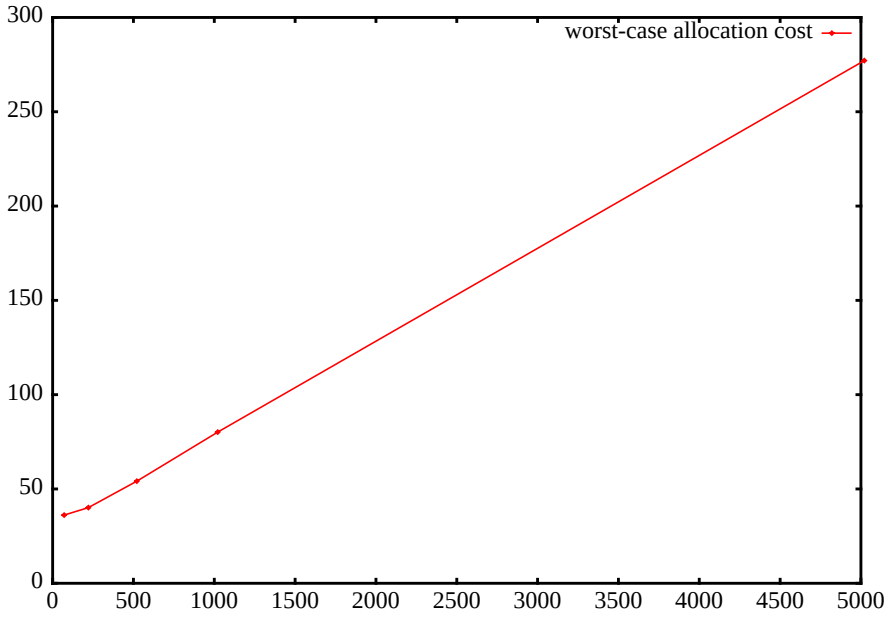


Figure 6.4 Worst-case allocation cost in microseconds as a function of object size (bytes).

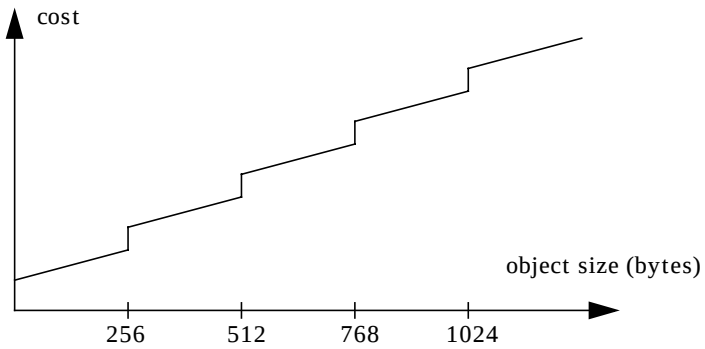


Figure 6.5 Theoretical cost of allocation requests made by a high-priority process. The cost is slightly saw-tooth shaped due to quantification effects caused by making object initialization interruptible.

essary to take a closer look at the implementation of the prototype. The anomaly is in fact a quantification effect caused by the requirement that object initialization must be interruptible. This is required in order to minimize the latency of high-priority processes. The initialization of an object is performed by two nested loops and executes mostly with the interrupts disabled. The inner loop is very tightly coded and initializes at most 256 bytes of memory each time it is invoked. The outer loop starts by calculating how much memory the inner loop is to initialize when invoked, i.e. the minimum of 256 and the amount of remaining memory to be initialized. Next, the inner loop is invoked. When the inner loop has finished, the interrupts are enabled and immediately thereafter disabled again. This allows a context switch to occur if any interrupt is pending. It thereby allows processes with higher priority to run. Running other processes might cause the garbage collector to move the object being initialized. Consequently, the pointers, located in processor registers, used for initializing the new object must be reloaded from safe locations before starting the next iteration of the outer loop. Altogether, there will thus be an additional cost after every 256 bytes of initialization. The cost function will therefore actually have the saw-tooth shaped form shown in Figure 6.5. All the object sizes used in the measurements happen to be located just before an additional iteration of the outer loop is required, except for the first one (50 bytes). This explains the irregularity observed in Figure 6.4.

Memory allocation in low-priority processes

Allocation requests made by low-priority processes cause GC work to be performed before a new block of memory is passed to the application. A large part of the worst-case cost for an allocation request will now consist of GC work. The SingleLP test

application was used to measure the allocation costs for objects of various sizes. The result is presented in Table 6.3.

50 bytes	*200 bytes*	*500 bytes*	*1000 bytes*
386 µs	447 µs	593 µs	799 µs

Table 6.3 Worst-case allocation times for low-priority processes when allocating objects of varying size.

Each test run used a total heap size of 100000 bytes and the maximum amount of live memory was 20000 bytes, which constitutes a fairly well-filled heap. We see that the allocation costs are reasonably small even for low-priority processes, despite the implementation of the GC algorithm being far from optimal. Sub-millisecond worst-case allocation times are achieved even for allocation of as large objects as 1000 bytes.

Response times

The difference between the SingleLP application and SingleHP is the priority of the process allocating memory. It is interesting to compare the response times of the process in the two cases. Table 6.4 reports the worst-case execution times for the SingleLP and SingleHP versions of the process when allocating objects of various sizes.

	50 bytes	*200 bytes*	*500 bytes*	*1000 bytes*
SingleLP	657 µs	722 µs	870 µs	1180 µs
SingleHP	323 µs	339 µs	372 µs	387 µs

Table 6.4 Worst-case execution times for the memory allocating process of the SingleLP and SingleHP test programs when allocating objects of varying size. The increase in response time for SingleHP with larger object size is due to memory initialization in connection with allocation. The response time would have been constant at around 320 µs if the initialization strategy described in Section 4.4.1 had been implemented.

When we compare the data for the two programs in Table 6.4 we see that the execution times for the SingleHP version are significantly shorter. However, if we calculate the total time for executing the application process and the GC process we find that the SingleHP application requires somewhat more CPU time than the SingleLP application. We thus buy shorter response times at the cost of an increased total CPU time requirement. The increase in total CPU time is explained

by the extra work involved with deferring the GC work and by the relatively high costs for managing context switches.

6.4.3 Allocation cost of manual memory management

The standard routines for memory management, *malloc* and *free*, of the real-time kernel were studied in order to get an idea how the costs of automatic memory management compare with those of manual memory management. The available implementation of malloc and free use a very simple algorithm. A single linked list is used to record all free blocks of memory. A first-fit strategy is used to decide where to allocate a new object. Blocks are joined together with adjacent free blocks when deallocated.

In our first experiment, we allocated objects with randomly chosen sizes in the interval 8 to 1000 bytes. When 2000 objects had been allocated, previously allocated objects were randomly selected and deallocated as new ones were allocated. The maximum and minimum costs of allocating an object (using malloc) and deallocating an object (using free) were measured. The results are displayed in Table 6.5.

	malloc	*free*
Max	150 µs	154 µs
Min	28 µs	44 µs

Table 6.5 Upper and lower costs for allocation when using manual memory management (malloc/free). Objects with sizes varying from 8 to 1000 bytes were allocated in a random order.

In the second experiment, we attempted to provoke malloc to display its worst-case behaviour. A free-list containing a varying number of free blocks was first artificially created. Then we attempted to allocate an object larger than any of the free blocks and measured the time required for the malloc call. The number of free blocks were varied from 50 to 5000. The results are summarized in Table 6.6

free blocks	*malloc*
50	483 µs
500	4.1 ms
5000	43 ms

Table 6.6 Worst-case allocation cost as a function of the length of the free-list used by the systems manual memory manager.

Table 6.6 illustrates that the existing malloc implementation is not very suitable for hard real-time systems since it is difficult to guarantee reasonably low response times when malloc is called. The cost of malloc is not predictable. Algorithms with a much better worst-case performance do exist (e.g. [Bre89]), but their best-case performance will hardly be better than the one we have experienced (the lower costs in Table 6.5). In fact, it is reasonable to expect that it will be somewhat higher due to their more complicated data structures.

The conclusion is that our garbage collector prototype does not cause more interference to high-priority processes than standard manual memory management strategies. In fact, it proves cheaper and more predictable in many cases. As an additional bonus, it delivers new objects with their contents initialized to predetermined values (zeroes), which manual memory management strategies typically do not. This decreases the risk of hard-to-find bugs related to programmers forgetting to initialize the values of the fields of new objects.

6.4.4 Latency for high-priority processes

The increase in latency for the high-priority processes induced by garbage collection is caused by the locking necessary to guarantee heap consistency in all situations. The locking is implemented by disabling/enabling processor interrupts. During the time the processor interrupts are disabled, no clock interrupt can occur and cause a context switch. Thus, determining the worst-case contribution from the garbage collector to the process latency is equivalent to determining the longest time the interrupts are disabled. All code that temporarily switch off the interrupts were therefore augmented with code producing a digital output signal with a duration equal to the time the interrupts were disabled. The signal was studied according to the method described in Section 6.2 and the results from running four of the test applications are presented in Table 6.7.

SingleHP	*GCTest*	*Pålsjö-Polynomial*	*Pålsjö-Robot*
61 µs (16 µs)	61 µs (7 µs)	56 µs (4 µs)	48 µs (4 µs)

Table 6.7 The worst-case locking times observed when running four of the test programs. The average-case locking times are presented within parenthesis.

We observe that the worst-case locking time is around 60 µs independent of the application. This is smaller than the cost of handling a clock interrupt and a subsequent context switch, which on the experimental system is in the order of 60+60 = 120 µs. The total latency, about 180 µs, should thus still be low enough for most hard real-time control applications. The worst-case locking time depends on the longest atomic operation in the memory manager. It is currently not known which piece of code stands for the longest atomic operation. By finding this piece of

code and tuning it to yield shorter locking time, the worst-case latency should decrease accordingly.

Invocation jitter is defined by variation in the time at which execution is handed over to a process with respect to the ideal invocation time. It is desirable that periodical high-priority control processes execute with very precise intervals. It is not always necessary that the latency is very small, as long as the variation of the latency, i.e. the jitter, is small [ÅW84]. Jitter is introduced by variations in the time necessary to handle a clock interrupt and perform a context switch. Also, a large source of jitter is processes with higher priority blocking the process. Garbage collection also contributes to the jitter by potentially adding time to the latency. Since locking caused by the garbage collector occurs at random occasions, all of the locking contributes to increasing the jitter. It should be noted, however, that most of the jitter for a process is caused by processes with higher priority and that this dwarfs the additional jitter caused by garbage collection. The only occasion where the GC-induced jitter could cause a problem is for a very critical control process with the highest priority of the system. We can imagine few systems where the current worst-case latency could not be tolerated, and in such a case, dynamic manual memory management would not be an alternative either.

6.4.5 Execution time for the garbage collector process

Garbage collection required by the actions of high-priority processes is deferred until such a time that no high-priority process is ready to execute. Then, processor control is turned over to a special GC process which performs the necessary GC work. Garbage collection motivated by the actions of low-priority processes is performed by the low-priority processes themselves.

The execution time for the GC process was studied in order to get an idea of what can be expected in control applications. We varied the amount of memory allocated by the SingleHP application each time its high-priority process was invoked. The worst-case execution time of the GC process varied from 414 µs to 787 µs when the size of the allocation requests varied from 50 bytes to 1000 bytes. The average-case execution times varied from 94 µs to 156 µs. It should be noted that our implementation is not very optimized since we have concentrated on finding techniques to minimize the response times for high-priority processes instead of implementing the GC algorithm as efficient as possible. Even so, the costs are comparable to what have been reported for other software-only real-time garbage collectors, e.g. [EV91].

6.5 Using the garbage collector in control applications

Two of the test programs used to test our garbage collector prototype were modified versions of existing control applications. Here we report on the consequences of using garbage collection in these applications.

6.5.1 Inverted pendulum control

The Pålsjö-Robot test application uses two periodic high-priority processes, $\tau 1$ and $\tau 2$, to implement the actual control. The sampling frequencies for the processes are 100 Hz and 40 Hz respectively. The control processes allocate objects containing plot information, which are then passed on to low-priority processes for presentation to the user.

Table 6.8 shows the worst-case execution times for the high-priority processes of the Pålsjö-Robot application. We ran the control processes with high priority as well as with low priority. In addition, we studied an unmodified version of Pålsjö, i.e. Pålsjö without automatic memory management. We see that the execution times are reduced with 0.1 and 0.2 ms respectively when the control processes are treated as high-priority processes instead of low-priority processes. Only a small part of the code of the control processes is related to memory management, which explains the relatively small difference in response time. Even so, the difference is significant. The worst-case cost of an individual allocation was reduced from 415 to 36 µs when the processes were made to run as high-priority processes. The GC process invoked when the high-priority control processes are suspended had a worst-case execution time of 303 µs and required about 1% of the total CPU time.

process	*high priority*	*low priority*	*no GC*
$\tau 1$	3.5 ms	3.7 ms	7.2 ms
$\tau 2$	5.8 ms	5.9 ms	9.6 ms

Table 6.8 Observed worst-case execution times for the control processes of the Pålsjö-Robot application.

The high execution times for the unmodified Pålsjö system, under the column titled *no GC* in Table 6.8, is interesting. It turns out that this is an effect of the difference in how the programmer chooses to write his code when automatic memory management is available. The modified Pålsjö system uses dynamically allocated objects to store plot data. References to the data objects are passed on to lower priority processes for presentation to the user. The original version of Pålsjö uses a more conservative approach to memory management in order to guarantee predictability and robustness. A number of statically allocated buffers are used to

store plot data. All the plot data is copied between buffers several times on its way from the control processes to the user. This explains the high execution times. It also illustrates how the introduction of automatic memory management often have a positive, but hard-to-quantify, effect on system performance. Programmers are free to use a wider range of implementation techniques when automatic memory management is available.

The CPU usage of the Pålsjö-Robot system is very high, around 97%. It is an example of a system that is very suitable for our semi-concurrent GC strategy. Preventing low-priority processes from allocating at a higher rate than the garbage collector can keep up with guarantees that critical processes are not delayed by garbage collection. Overload affects primarily low-priority processes. Only very heavy overload can prevent the garbage collector from keeping up with the high-priority processes.

6.5.2 Polynomial regulator

The Pålsjö-Polynomial application implements an adaptive controller for a servo motor. The system contains two critical processes, $\tau 1$ and $\tau 2$, with sampling frequencies 50 Hz and 10 Hz respectively. Process $\tau 1$ implements the actual control of the servo whereas process $\tau 2$ continuously monitors the regulators performance and adaptively updates the control law used by $\tau 1$. The processes uses dynamically allocated objects to represent polynomials and temporary objects are used frequently. In addition, the processes generate plot data which is passed on to lower-priority processes in dynamically allocated objects. The system is quite CPU intensive with a total CPU usage of about 75%. The two control processes contribute to the majority of the CPU load.

The execution times for the two processes were studied. We measured the worst-case execution times when the control processes were executed as high-priority processes as well as when they ran as low-priority processes, see Table 6.9. We do not present any data for a variant of Pålsjö-Polynomial using manual memory management, since we felt that the existing implementations differed to much to make a comparison meaningful. We expect that the result of such a comparison, if it was feasible, would yield similar results as it did for the Pålsjö-Robot program.

process	*high priority*	*low priority*
$\tau 1$	8.9 ms	9.6 ms
$\tau 2$	19.5 ms	22.7 ms

Table 6.9 Observed worst-case execution times for the control processes of the Pålsjö-Polynomial application.

We see that the response times of the control processes were reduced significantly when treated as high-priority processes due to lower allocation costs. The worst-case cost for an individual allocation request was reduced from 598 to 63 µs by changing the priority of the processes. The execution time of the GC process, cleaning up after $\tau 1$ and $\tau 2$, never exceeded 2.06 ms. The GC process required about 1.5% of the total amount of available CPU time. A 200 kilobyte heap was used for all measurements and the maximum amount of simultaneously used memory was 30 kilobyte.

6.6 Summary

The experiments described in this chapter verify that we have achieved the goal of reducing the worst-case cost for automatic memory management for critical control processes to such a level that it can be used even in hard real-time applications. The current garbage collector implementation, based on an incremental copying algorithm and running on a 25 MHz Motorola 68040 microprocessor, incurs an overhead consisting of a single machine instruction for pointer dereferencing. The worst-case cost for a pointer assignment was found to be 9 µs in the applications used in our experiments. The worst-case cost of memory allocation in high-priority processes is currently dependent on the size of the requested memory block since the contents of the new object are initialized at allocation time. Our measurements indicate, however, that the worst-case allocation cost will be below 30 µs once the memory initialization strategy described in Section 4.4.1 is implemented.

Our prototype garbage collector has been used in actual control applications with encouraging result. The response times of high-priority control processes could be reduced by using semi-concurrent garbage collection while still preserving predictability. This was especially evident in the inverted pendulum experiment, in which the reduction was in the order of 0.1 ms. In the inverted pendulum experiment, the system is very close to overload and the lowest-priority processes frequently miss their (soft) deadlines. Still, the schedulability of the high-priority processes can be guaranteed. Interleaving garbage collection with mutator execution, performing the GC work in connection with memory allocation, would here cause extended response times and fully concurrent garbage collection would not be able to guarantee high-enough GC rates to meet the allocation needs of the high-priority processes.

The aim of the implementation work was merely to prove the feasibility of the proposed GC scheduling strategy, not to achieve best overall performance or user friendliness. We believe that the goal of the work has been reached, even though some obvious opportunities for improvement exist:

- The overhead for object initialization should be removed from memory allocation in high-priority processes, making the worst-case cost for allocation small and independent of object size.

- The response times for high-priority processes can be reduced further by using GC algorithms with cheaper read/write barriers. An example of such an algorithm is incremental compacting mark-sweep, described in Section 4.9. Non-moving GC algorithms can be an alternative in very demanding applications.
- GC algorithms with lower overall overhead can be used in order to improve the performance of, primarily, the low-priority part of the system. Here, generation-based algorithms are of special interest.
- The garbage collector should be integrated with the language system used for implementing real-time applications. This would relieve the programmer from having to follow all the detailed coding conventions imposed by the current library-based garbage collector and ensure robust and safe application programs.

The above issues should be addressed in the next version of the garbage collector prototype. We plan to move to a PowerPC-based platform which should improve performance drastically. It seems that the PowerPC microprocessor will be a popular choice for use in future embedded systems.

Chapter 7

Related Work

After having described our approach for garbage collection in hard real-time systems, we are now ready to compare it with previously published techniques. This chapter surveys some of the most important approaches and relates them to the work presented in earlier chapters.

7.1 Incremental copying algorithms

Even though incremental compacting mark-sweep algorithms exist, as described in Section 4.9 and [Ben90], copying algorithms dominate the literature. Several of these have been labelled real-time algorithms, but they generally provide only soft real-time performance if implemented without modifications to how their work is scheduled.

7.1.1 Baker's algorithm

The best-known incremental copying algorithms is Baker's algorithm [Bak78], which is a precursor to the algorithm we use in our garbage collector prototype. It was denoted a real-time algorithm since it guarantees low upper bounds on the cost of every pointer operation.

A read barrier is used to ensure that the mutator only sees grey or black objects, according to the tri-colour marking terminology described in Section 4.3.1. This prevents the mutator from inserting pointers to white objects into objects already scanned by the collector. Any pointer access might thus require that an object is converted from white to grey. Turning an object grey implies copying it from one semispace to another. The worst-case cost of performing a piece of code containing several pointer accesses will therefore be very high, since every access might trigger copying.

The work of the garbage collector is performed in small increments triggered by allocation requests. Allocating an object thus means that the mutator is suspended for a short period of time while GC work is performed. The length of a GC increment is not very predictable, however. This is partly due to a too simple way of measuring GC work, see Section 4.7, and partly due to the high cost of performing a *flip*. A flip initiates a new GC cycle and, as the algorithm was originally formulated, requires that all root pointers of the system are scanned in a single atomic operation. This might take a very long time since it will typically imply that a number of objects must be copied. These deficiencies add to making worst-case response times very long for high-priority processes.

To summarize, the high worst-case costs for pointer access and object allocation make Baker's algorithm unsuitable for hard real-time systems, since it is very difficult to guarantee that hard deadlines are always met. This could be alleviated by changing how the GC work is scheduled, for example by using the scheduling strategy proposed in Chapter 4 of this thesis. The read barrier would also need to be modified. Lazy copying would be required in order to minimize the cost of pointer accesses. Also, scanning all the root pointers cannot be performed in one single atomic operation, but must be handled incrementally.

7.1.2 Brook's algorithm

Brook's algorithm [Bro84] is a development of Baker's algorithm using a write barrier instead of a read barrier. This makes the overhead smaller since pointer writes are much less frequent than pointer reads. Otherwise, it has essentially the same real-time properties as Baker's algorithm. It too requires that all root pointers are scanned atomically in connection with a flip. Bengtsson [Ben90] has shown how scanning the root pointer set can be made incrementally, solving this problem.

The algorithm we use to illustrate our scheduling principle is based on Brook's algorithm with Bengtsson's modification. Lazy copying is used to minimize the cost of the write barrier (see Chapter 4).

Another example use of Brook's algorithm can be found in the Erlang system [Vir95]. The variant of the algorithm used there is similar in many respects to how we perform garbage collection for low-priority processes. The root pointer set is scanned incrementally in order to avoid long delays in connection with semi-space flips and the metric for performed GC work is almost identical to ours (Section 5.2.6). A difference between our collector and the Erlang collector is that we calculate a static GC rate whereas the Erlang collector is self-adjusting in this aspect. A consequence of this is that the Erlang system cannot a priori guarantee that hard deadlines are met in every situation. Since the language is primarily intended for soft real-time applications (telecom), the inability to meet hard deadlines is of minor importance.

7.1.3 The Appel-Ellis-Li collector

Virtual memory hardware is used to implement a read barrier in the Appel-Ellis-Li garbage collector [AEL88]. The mutator is only allowed to see black object, i.e. objects that are both evacuated and scanned. Attempts to access unscanned (grey) objects in tospace generate a page access trap. The trap causes the collector to scan all objects located on that page, thus turning them into black objects. The collector can run concurrently with the mutator, tracing and evacuating live objects. The advantage of the scheme is the low cost of the read barrier as long as only black objects are accessed. However, the worst-case cost is very high since every pointer access might result in a trap. Each trap is costly since all objects on the page must be scanned, which might cause many objects to be copied from fromspace into tospace.

7.1.4 Real-time replication garbage collection

An interesting variant of copying collection is replicating garbage collection, suggested by Nettles and O'Toole [NO93]. Most other algorithms maintain a *tospace* invariant, i.e. some kind of read barrier (in some cases as simple as using a forwarding pointer) ensures that the mutator always accesses the tospace versions of objects. Nettles and O'Toole instead enforce a *fromspace* invariant, meaning that the mutator always accesses the fromspace copies. No read barrier is required. A write barrier is needed to record writes to objects that have been evacuated to tospace in a mutation log. The collector concurrently evacuates objects and updates already evacuated objects according to the contents of the mutation log. A flip is performed when all live objects have been evacuated from fromspace and the mutation log is empty.

The advantage of replicating garbage collection, according to Nettles and O'Toole, is the low cost for pointer access and pointer write. The collector can easily run concurrently. The amount of work that has to be performed before a flip depends to a high degree on the mutation pattern of already evacuated objects, making it unpredictable. We thus conclude that the algorithm is not directly applicable to hard real-time systems.

7.2 Non-moving garbage collection

One approach to making garbage collection feasible in hard real-time systems is to give up memory compaction. This has several advantages. The total amount of necessary GC work decreases since objects do not have to be moved. The cost of operations on pointers, such as pointer dereferencing and pointer assignment, will typically be smaller. The idea is to make the worst-case costs of these operations small enough not to cause the application to miss any hard deadlines. It is also eas-

ier to synchronize the mutator and the collector since the collector never modifies pointers visible to the mutator.

The disadvantage of non-moving garbage collection is memory fragmentation. This in turn means that the worst-case memory requirement may be very large [Rob71]. In addition, free memory must be registered in some form of free-list, potentially making memory allocation unpredictable, even though very efficient schemes for organizing and managing the free-list have been reported [Bre89].

7.2.1 The Treadmill

Baker has proposed an incremental non-moving algorithm, the Treadmill [Bak92], that works much along the same principles as his copying algorithm. All objects, dead as well as alive, are linked together in a circular double-linked list. The circular structure is organized into four segments, denoted white, grey, black, and free in the order they occur in the list. The collection status of an object, black, grey, white, or belonging to a free-list, is determined by which segment it is located in. Pointers are used to keep track of where each segment starts. Individual memory management operations become very cheap using this optimization. Marking an object as scanned can be done by moving the scan pointer one step forward in the list, effectively turning a grey object into a black one. If all objects are equally sized, allocations can be performed just as easy. Otherwise, a search through a freelist must be performed. Turning an object grey involves relinking it into the grey section, but this is a cheap operation with constant cost.

The algorithm does have some problems of handling different-sized objects, resulting in memory fragmentation and non-deterministic allocation times. Wilson and Johnstone [WJ93] have implemented a variant of Baker's Treadmill which solves the problem with unpredictable allocation times. They round object sizes up to the nearest power of two and uses separate treadmills for each size class. The fragmentation problem is, however, not solved.

7.2.2 Yuasa's algorithm

The algorithm published by Yuasa [Yua90] is an example of a non-moving mark-sweep garbage collector. It uses a "snapshot-at-the-beginning" strategy in order to guarantee that the mutator does not modify the pointer graph behind the back of the collector in a dangerous way. This means that all objects reachable by the application at the beginning of a GC cycle will be considered live and will be retained until the next cycle. A write barrier is used to enforce the snapshot-at-the-beginning strategy. If a pointer write would overwrite a pointer to an object not yet identified as live by the garbage collector, i.e. a white object in the tri-colour marking terminology, the pointer to be overwritten is first pushed onto a GC stack. The GC stack is used by the collector to store pointers to objects identified as live, but

not yet traversed, i.e. grey objects. All objects reachable at the start of a GC cycle will thus be identified as live, either by the collector or by the write barrier of the mutator. Compare this with the *incremental-update* strategy employed by Baker's algorithm among others. There, a read or write barrier ensures that new pointer values do not point to white objects, thus preventing disruptive pointer assignments behind the back of the garbage collector.

The worst-case costs of common pointer operations are very low for Yuasa's algorithm. No read barrier is necessary and there is consequently no extra overhead for pointer access. A write barrier is necessary, however, but even here the cost is very low. The write barrier will push the old value of the target pointer onto a stack and possibly update a mark bit in the object referenced by the old pointer value. This is a small and constant worst-case cost. Pointer reads and writes should therefore not pose any threat to meeting hard real-time deadlines.

Allocation requests can, however, be quite costly. Yuasa uses the same strategy as Baker [Bak78] and many others to schedule the work of the garbage collector. An increment of GC work is performed in connection with each allocation operation, meanwhile suspending the mutator. Enough work is performed to guarantee that the garbage collector keeps up with the allocation requests of the mutator. It is true that the worst-case costs for individual memory management operations are bounded by reasonably small values, but a high-priority process emitting several allocation requests might even so be delayed long enough to fail to meet its deadline in a demanding application.

A way to fix the problem with long delays in connection with allocation requests would be to change the scheduling strategy. Instead of performing an amount of GC work in connection with each allocation request, the work of the garbage collector could be scheduled in the same way as was proposed in Section 4. Allocation requests by high-priority processes would no longer be associated with high worst-case costs. The missing GC work would be performed in the pauses in between executing high-priority processes. Yuasa's algorithm, scheduled according to the principles described in this thesis, thus seems to be a suitable garbage collector for hard real-time systems, provided that memory fragmentation can be accepted.

7.3 Hardware-supported garbage collection

One way to make the worst-case overhead for pointer operations (such as pointer access, pointer assignment, and allocation) small enough to be acceptable in hard real-time systems is to employ specialized hardware. This is the approach used by Nilsen and Schmidt in their hardware-assisted garbage collector [NS94]. They have constructed a special memory module that interfaces to the processor via the conventional system bus. The memory module implements a heap which is garbage collected using a variant of Baker's algorithm [Bak78]. It is essentially a two-processor approach to garbage collection since a separate processor embedded in the memory module is responsible for performing garbage collection. Historically, spe-

cialized computer hardware has seldom proved to be successful. The problems are that it is difficult to keep up with the rapid processor development and that the hardware is not sold in enough volumes to be profitable. It is hoped that the general design of the memory module will allow it to be used in a wide range of existing and future architectures, thus making it a viable solution.

We are not aware of any existing performance data for the hardware-assisted garbage collector derived from an actual implementation. Only data produced by simulations seem to be published. It is unclear on what hardware performance equivalent to the results of the simulation could be expected. Nilsen and Schmidt refer to 50 MHz CPU:s and standard workstations (of the year 1994) in their papers so it seems reasonable to assume that this is the kind of hardware they have in mind.

It is claimed that pointer operations and memory allocation take at most 1µs to perform. It is admitted that a memory allocation request will take a considerably longer time to perform if the garbage collector has to perform a flip, namely in the order of 500 µs during which time the mutator is suspended. The long delay is due to the fact that the CPU cache must be flushed. Even though flips will occur only with long intervals (typically in the order of once every tenth or twentieth second) this can seriously affect the ability of a very time-critical process to meet its deadline.

The normal 1 µs worst-case cost for allocating an object can be questioned. It appears that this is only true if the garbage collector of the memory module is able to keep up with the memory allocation rate of the mutator. If the mutator allocates memory at a very high rate, the memory module will stall the central processor until it has performed enough GC work to keep up with the allocations. A conceivable scenario is a low-priority process that allocates a series of large objects being preempted by a time-critical high-priority process. The high-priority process makes an allocation request which causes the memory module to stall the central processor. The worst-case duration of such a stall has not been studied as far as we know. In order to guarantee that such stalls never occur, a detailed scheduling and memory management analysis of the entire system would be required. The described scenario is not a problem in our approach since low-priority allocation will never cause the garbage collector to get behind with its work. Only a limited analysis of the behaviour of the high-priority processes is required in our case in order to guarantee that the high-priority processes will never be delayed by garbage collection.

It is interesting to compare the worst-case costs of the hardware-assisted garbage collector with the worst-case costs for high-priority processes when using our garbage collector prototype. We achieve sub-microsecond worst-case costs for pointer accesses and a maximum of 10 µs for pointer assignments on a 25 MHz Motorola 68040 microprocessor. We expect that the latter cost will decrease further when using GC algorithms that can use a simpler write barrier. Even though our prototype has lower worst-case allocation costs than the hardware-assisted gar-

bage collector, allocations are usually more expensive in our prototype. A large part of our overhead is due to the fact that we currently initialize the contents of new objects at allocation time, whereas the hardware-assisted garbage collector does consider this to be a part of the work of the garbage collector. The memory is thus already initialized when an allocation request is made. We will use the same technique in future versions of our prototype garbage collector, which will significantly reduce the cost for memory allocation in high-priority processes.

7.4 Concurrent garbage collection

Many real-time GC algorithms schedule their work proportional to the amount of newly allocated memory. An example of such an algorithm is Baker's algorithm [Bak78]. Enough work is performed, often in connection with allocations, to guarantee that the mutator will not run out of memory. When this is done, garbage collection is suspended. The GC work will thus disrupt the execution of the mutator. A solution to this is to perform garbage collection in a separate thread. In this way, the mutator thread is normally not disrupted by garbage collection and otherwise idle time can be spent on garbage collection. The collector thread must be able to keep up with the allocation requests made by the mutator. Otherwise, the mutator must be suspended until the collector has caught up.

Steele [Ste75] proposed concurrent garbage collection as a means to remove the disruptive pauses caused by stop-the-world garbage collection. He showed how to synchronize the work of a mark-sweep collector with the mutator. The intention was that the collector and mutator would execute on separate processors, but a time-slicing approach for single-processor systems was also discussed. The aim was not to support real-time computing and the work does not provide upper bounds on the cost of pointer operations and memory allocation. The work could be considered to be a forerunner to later incremental algorithms.

Dijkstra & al. [DLM+78] is another example of an early proposition for concurrent garbage collection. Its properties are similar to those of Steele's strategy, but more focus was given to minimizing exclusion and synchronization constraints.

Appel-Ellis-Li [AEL88], see also Section 7.1.3, perform copying garbage collection in parallel with executing the mutator. They use virtual memory hardware to implement a read barrier variant. The read barrier ensures that the mutator only sees tospace pointers. Any access to an unscanned page, i.e. a page that may contain fromspace pointers, causes a page-access trap. The entire page is then scanned, while the mutator is halted. Every pointer access would cause a page-access trap in the worst case. The authors reports that the cost for an individual trap is in the order of 40 ms. Even though the time for a trap can be reduced [Joh92], the worst-case costs are still too high for use in hard real-time systems.

Boehm, Demers, and Schenkers [BDS91] describe something called "mostly parallel garbage collection". Here, the mutator is allowed to execute concurrently with the collector tracing the set of live objects. The algorithm relies on virtual

memory support to set *virtual dirty bits* when the mutator writes to a page of virtual memory. The objects in the dirty pages are scanned in a stop-the-world way when the collector has finished its concurrent traversal of the pointer graph. No real-time demands can be guaranteed to be met.

Nilsen and Schmidt [NS94] rely on concurrent garbage collection. Here, the GC work is performed by an embedded processor in a specialized hardware memory module. There is no easy way to guarantee that high-priority processes are never disrupted for extended periods. The memory module will stall the central processor if it does not keep up with the current allocation rate. This approach is described in more detail in Section 7.3.

Summary of concurrent collection

A common property of all the described approaches is that they attempt to perform all GC work concurrently with the mutator. This leads to difficulties when it comes to guaranteeing that a real-time application meets all its hard deadlines. It must be proved that time-critical high-priority processes are never delayed for too long periods of time by garbage collection. Since the actions of each and every part of the application program may cause the collector to stall the mutator, a scheduling and memory management analysis of the complete program must be performed.

In our approach, time-critical high-priority processes are treated separately. Garbage collection is performed concurrently in respect to high-priority processes, but is interleaved with the execution of the mutator for low-priority processes. Low-priority processes are never allowed to allocate memory without performing the corresponding GC work. Consequently, only allocation requests made by a limited number of high-priority processes can cause the garbage collector thread to get behind with its work. We therefore only have to analyse the behaviour of the high-priority processes in order to guarantee schedulability.

7.5 Special treatment of high-priority processes

An approach similar to the one described in this thesis was presented by Withington in a short paper [Wit92]. It describes a proposed GC scheme for a new Lisp real-time kernel developed at Symbolics Inc. The new kernel is a development of the Genera operating system from the same company.

Genera uses a variant of Baker's algorithm (described by Moon [Moo84]), but this was deemed unsuitable for hard real-time systems because of the arbitrary delays caused by the read barrier. The cumulative delay would be too large to guarantee that high-priority processes would always meet their deadlines. The proposed solution is based on the same basic idea as the work described in this thesis; defer the work motivated by garbage collection until the high-priority process is suspended. Processes with high and low priority are thus handled differently.

However, only the work required by read and write barriers was proposed to be deferred. Allocation requests would still cause an amount of GC work proportional to the size of the requested object to be performed.

Our work differs from Withington's proposal in several ways. We too propose that GC work should be deferred until such a time that no high-priority process is executing, but we defer *all* GC work, including the work motivated by memory allocations. We use identical read/write barriers for high and low priority processes, eliminating costs for deciding which variant to use in each case. We also provide means to perform schedulability analysis in order to show that a system is schedulable, which Withington's paper does not. It is only noted that the strategy depends on enough CPU time being available to clean up after the high-priority processes.

We are not aware of any papers or other kinds of publications expanding on the proposed garbage collector technique or presenting any experience from using it.

7.6 Summary

Most of the previous work within the field of garbage collection for hard real-time systems has focused on producing very fine-grained incremental GC algorithms, with good results. The scheduling issue has not been as thorough studied, however.

Two major approaches to scheduling the GC work have been proposed. The first approach is to schedule small increments of GC work in connection with memory management operations performed by the mutator. The mutator and the collector are strongly synchronized. This has the advantage that it is easy to show that the collector will keep up with the allocation requests by the mutator, but it also implies that the mutator will suffer from frequent GC-induced pauses. The accumulative pauses can quickly make it impossible to guarantee that a high-priority process will always meet a tight deadline. The second approach to GC scheduling is to perform GC work asynchronously as a separate concurrent process with lower priority than the mutator processes. By doing so, high-priority processes will no longer suffer from GC-induced pauses. On the other hand, it is now more difficult to prove that the collector will always be able to keep up with the mutator. An extensive scheduling analysis including every mutator process in the system is required to do so.

Our approach differs from most previous approaches by combining the sequential and concurrent approaches in order to take advantage of the good properties of both. GC work is scheduled concurrently with high-priority processes avoiding GC-induced pauses for these processes, while it is interleaved with the execution of the low-priority processes. Only a limited scheduling analysis, only incorporating the high-priority processes, is necessary in order to guarantee that the collector will keep up with the mutator. We further show how to do the scheduling analysis, an issue that no other work on garbage collection has addressed.

Chapter 8

Future Work

This thesis shows that garbage collection, with compaction, is a viable alternative for memory management even in systems that have to comply with hard real-time demands. Our experience shows that the strategy for scheduling the GC work is important in order to meet hard real-time requirements and we have therefore developed a suitable strategy.

However, some work and maturity remain before garbage collection is a standard component of commercial real-time kernels. The work presented in this thesis can be followed up in at least two main directions. The first direction is to further improve implementation techniques and make larger case studies. The second direction involves improving the tools and methods needed to analyse the schedulability of hard real-time applications.

8.1 Implementation

The current garbage collector prototype is implemented in a quite straight-forward way. The primary purpose of the prototype was only to prove the feasibility of the scheduling strategy proposed in this thesis. Although useful as it is, future incorporation into real-time products requires that substantial work on implementation issues is carried out.

Memory initialization

The contents of newly allocated objects must be initialized in order to guarantee that all pointers within the objects contain consistent values. The current implementation initializes an object by writing zeroes into it as a part of the allocation operation. This means that it is the *mutator* that is responsible for object initialization. A consequence of this is that the cost for allocating an object in a high-

priority process will depend on the size of the new object. Object initialization stands for the major part of the total cost. The alternative strategy for memory initialization described in Section 4.4.1 should be implemented. This makes the worst-case allocation cost for high-priority processes small and independent of object size.

Case studies

The case studies made this far are somewhat limited. The application programs used in the studies have only been partially converted to using automatic memory management. Even so, they have provided valuable data. We have been able to derive the costs for individual memory management operations, such as allocation, and experienced that automatic memory management can be used in a demanding control application without violating hard real-time demands.

A key motivation for using automatic memory management in embedded systems is that it is claimed that programs using automatic memory management are easier do design, implement, and debug. This in turn leads to safer and more robust products that are cheaper to implement and easier to maintain. However, it still remains to demonstrate that these claims are indeed true for embedded systems. Further case studies are necessary in which automatic memory management is used more aggressively.

Mark-sweep compacting algorithms

The current prototype garbage collector implements a modified variant of Brook's incremental copying algorithm. The decision to use this algorithm was based more on good previous experience with the algorithm than on a systematic evaluation of different candidate algorithms. There is nothing that indicates that a copying algorithm is the best choice for hard real-time systems. On the contrary, it could be argued that a compacting mark-sweep type collector has properties that would make it more suitable. For example, the memory footprint of a mark-sweep collector will probably be smaller than that of a copying algorithm. Also, mark-sweep algorithms promise better performance when the ratio of live memory is high [Ben90]. These properties are probably desirable in embedded systems where the available memory is limited and the ratio of live memory is high.

The performance of a compacting mark-sweep algorithm scheduled according to the principles described in this thesis should be investigated. The impact, if any, of the algorithm change on the scheduling analysis should be determined. A prototype implementation should be made according to the discussion in Section 4.9 and compared with the existing prototype.

Improve average-case performance

Generation-based garbage collection is a well-known technique to achieve good average-case performance and good overall efficiency. We have observed that a garbage collector that never performs any work while high-priority processes are running can to a higher degree give priority to achieving good average-case performance than other collectors, without violating hard deadlines. A generation-based algorithm therefore appears to be suitable for a collector based on our scheduling strategy, as described in Section 4.10. Using a generation-based algorithm will not improve the situation for the high-priority processes, but it will probably produce better average-case performance for the low-priority processes that do not have to comply with hard deadlines. This should be verified in practice. An implementation of such a garbage collector should be made and its performance should be compared with that of single-generation garbage collectors. This direction of future work will probably be closely coupled with the investigation of the performance of mark-sweep GC algorithms, since such algorithms are obvious candidates for use in the older generations.

Language integration

Currently we use a library-based implementation technique for the garbage collector. A set of macros and functions is used by the application program to interface with the garbage collector. A large number of programming conventions must be followed by the application program in order to guarantee correct memory management, see Section 5.3. This conflicts with the very intention of introducing automatic memory management; namely reducing code complexity and improving robustness.

For a prototype implementation, whose purpose is to test the scheduling strategy and various implementation techniques, a library-based garbage collector is appropriate and sufficient. It is not sufficient when it comes to implementing a production-quality development environment, however. A development system is required which ensures correct memory management in a way that is transparent to the programmer. The compiler should automatically emit code compatible with the garbage collector and provide object layout information. A safe language, such as Java, would be required. Clearly, C and C++ are not suitable languages due to their inherent lack of type safety. Our plans for the future include developing a real-time Java compiler capable of meeting the hard real-time demands of embedded systems. It will be used both for studying what benefits one get from using Java and object orientation in control systems and as a platform for future GC research.

Completely avoiding the use of unsafe languages in embedded systems is probably not possible. Small routines coded in C or assembler will probably always be required in order to interface with hardware and to achieve extreme performance. The programmer will probably always be solely responsible for managing the

memory in a correct way in such routines, but he will be relieved from this for the bulk of the application code written in higher-level languages. The language issues should be investigated further and a full-scale development system should be developed.

8.2 Analysis

It is important that the worst-case performance of critical embedded systems can be derived before actually running the system in order to avoid expensive and potentially fatal failures. The tool for doing this is called a priori scheduling analysis.

Generalized analysis

The process model used in the scheduling analysis presented in Section 4.8 is somewhat limited. It assumes that high-priority processes execute periodically, which is true in most embedded systems. It is implied that the scheduling analysis can be extended to asynchronous processes as well, but the issue has not been investigated in detail. It should be demonstrated how the existing theory for rate monotonic analysis can be used to prove schedulability even in this case. Process scheduling techniques other than fixed-priority scheduling, such as earliest deadline first, could also be studied and it could be investigated how to analyse the schedulability of garbage collection in such a system.

Determining worst-case execution times

The schedulability analysis presented in this thesis assumes that worst-case execution times are available for the high-priority processes and for the GC work motivated by allocations performed by the high-priority processes. A major problem with this is that worst-case execution times may be hard to determine. This is a problem that our scheduling analysis shares with practically all other work on scheduling analysis. Sometimes the target computer system is predictable enough and the application processes simple enough to make it possible to calculate the worst-case execution time by analysing the program code, but in most cases we have to resort to more heuristic methods. One often employed method is to run the program and monitor the execution time of the various processes and from that estimating the worst-case execution times. We have to remember, however, that this method generally produces an optimistic estimate. There will in most practical cases be a risk that we sometime in the future will encounter a longer execution time than what we have experienced while monitoring the system.

The introduction of garbage collection complicates the issue somewhat. Not only do we require the worst-case execution time for each high-priority process, we

must also determine how much time the garbage collector will require in the worst case to clean up after the high-priority processes. Another way to put it is that we must determine the worst-case GC time as a function of how much memory a high-priority process allocates. This requires either detailed knowledge of how the garbage collector is implemented or some way of measuring the worst-case GC time (if we choose a heuristic approach). Again, the problem of determining worst-case execution times is a general problem for all approaches to schedulability analysis and must be considered to fall out of scope for this thesis, but the additional complications associated with introducing garbage collection should be studied further.

Improved estimation of performed GC work

As was described in Section 4.7, we must be able to make a good estimation of how much GC work has been performed in order to decide how much work to do in response to each allocation request. We would like to be able to determine the maximum *time* required to perform the GC work of one GC cycle. When the garbage collector is invoked after high-priority processes have been running it would be allowed to run for a *time* proportional to the amount of memory allocated by the high-priority processes. Since the hardware found in embedded systems seldom (never) includes a real-time clock with sufficient resolution, i.e. at least microsecond resolution, we have to resort to other methods of estimating performed GC work. However, we want the estimation to correlate with the time spent on GC work as well as possible. Our scheduling analysis assumes that we know the time required for GC work after a high-priority process has executed. The time actually spent on GC work must therefore be bounded, predictable, and small, even though we can not use a timer to abort it.

More work should be devoted to the problem of making good estimations of how much GC work has been performed. The estimation should correlate well to the time spent on GC work and at the same time not require to much work by itself. Some kind of compromise must be made between these two demands. It should be studied how to make such a compromise. Poor correlation between actual time spent on GC work and estimated work will result in unnecessary long worst-case estimations of the time required for garbage collection work. It should be analysed what impact this has on the performance of a garbage collector.

Chapter 9

Conclusions

This thesis deals with embedded control systems and the issue of efficient and robust memory management in such systems. We have concluded that automatic memory management, i.e. garbage collection, is desirable in embedded systems. The difficulties associated with introducing garbage collection without violating any hard timing constraints have traditionally impeded its use in hard real-time systems. We have presented a strategy for how automatic memory management can be made feasible in embedded control systems.

9.1 Contributions

A major contribution of this thesis is that it ties together research within the field of garbage collection algorithms with research in schedulability analysis. We have pointed out the need for efficient GC algorithms as well as for methods to analyse systems using garbage collection in order to improve the robustness and scalability of safety-critical control systems.

To achieve the above, we have presented techniques for scheduling the work of existing sequential incremental garbage collection algorithms in such a way that safety-critical processes are not disturbed. We have also shown how a priori scheduling analysis can be applied to control systems using automatic memory management. Furthermore, hands-on experience from implementing a garbage collector according to the principles described in the thesis has been presented.

Scheduling strategy

We have produced a garbage collection scheduling strategy suitable for embedded control systems by taking advantage of the inherent properties of such systems. The safety-critical parts of control systems are usually isolated to a few periodical-

ly executing high-priority processes. We have presented a scheduling strategy that defers virtually all GC work until such a time when no high-priority process is ready to execute. The costs for pointer access, pointer assignment, and memory allocation are small and bounded for high-priority processes. Given knowledge about the execution pattern of the high-priority processes, it is possible to guarantee that enough free memory is always available to meet the allocation requests of the critical processes. The GC work required to keep up with the allocation requests made by the high-priority processes is performed by a concurrently executing process. GC work is also performed interleaved with the execution of low-priority processes. Low-priority processes perform an increment of GC work when an allocation request is made. This has the effect that large requests for new memory made by low-priority processes can never affect the systems ability to meet allocation requests made by high-priority processes. We have shown how to combine the advantages of concurrent GC (for high-priority processes) with the advantages of sequential GC (for low-priority processes) while avoiding their respective disadvantages.

Schedulability analysis

Previous work within the field of garbage collection for hard real-time systems has primarily been focused on producing increasingly more efficient GC algorithms with shorter worst-case delays. Such work is very important, given the high CPU loads and tight deadlines found in modern control systems, but it must also be possible to determine whether a given application will meet all its hard deadlines or not. That is, techniques for a priori scheduling analysis must be available to the developers.

We have demonstrated how a standard technique for schedulability analysis, i.e. rate monotonic analysis, can be applied to control systems with automatic memory management. Rate monotonic scheduling is a very wide-spread method for process scheduling in embedded applications, which implies that the presented results are valid for a wide range of industrial applications.

Performing schedulability analysis in a system with garbage collection requires some extra information. Not only must the execution patterns, worst-case execution times, and deadlines for each process be known, the worst-case memory allocation needs of the processes must be taken into account as well. Without garbage collection, we typically only make sure that every process always finishes before its deadline, but here we also have to make sure that free memory is always available at the right time. This requires some additions to standard rate monotonic analysis.

An attractive property of our scheduling strategy is that an analysis of the schedulability of the high-priority processes does not have to take the actions of lower priority processes into account, except for blocking caused by access to shared resources. Allocation requests made by low-priority processes are never

allowed to prevent high-priority processes from allocating memory. Thus, low-priority processes with soft deadlines can be excluded from the analysis. This simplifies the analysis considerable since we only require detailed information about a small set of processes, namely the high-priority ones. Blocking by low-priority processes are handled using priority inheritance schemes, which can easily be incorporated in the analysis.

Implementation

We have developed a prototype garbage collector in order to experimentally verify the scheduling techniques proposed in this thesis. An existing real-time kernel was supplemented with a garbage collector and the performance of the garbage collector was evaluated.

The results of the experiment show that it is possible to remove practically all garbage collection overhead from critical high-priority processes. The worst-case costs of pointer operations and memory allocation are small and predictable. With the hardware used in the experiments, a pointer assignment takes less than 10 μs in the worst case. This can be further reduced by exchanging the copying GC algorithm used in the prototype for an algorithm with a cheaper write barrier, e.g. an incremental mark-sweep algorithm. The cost of allocation consists of a small constant time (less than 30 μs) for allocating an object plus a time proportional to the size of the requested object to initialize the contents of the new object. The latter cost component can be eliminated if the garbage collector is made responsible for memory initialization.

9.2 Consequences

The techniques for performing garbage collection in hard real-time systems have matured considerably during the last decade. We have contributed to this by showing how to use the properties of the target systems to schedule the work of the garbage collector such that it does not disturb safety-critical processes. The worst-case cost of individual memory management operations have been reduced to 10-20 machine instructions. We have also stressed the importance of scheduling analysis capable of handling systems with automatic memory management and provided methods for the analysis. It will probably not be long before automatic memory management makes its break-through within the field of real-time computing.

The introduction of automatic memory management in real-time systems has several benefits:

- Automatic memory management yields safer and more robust software than manual memory management techniques do. The problems with dangling pointers and memory leaks are eliminated. Much error-prone code administrating memory can be removed from the application program.
- Memory fragmentation can be avoided by using compacting GC algorithms. Without memory fragmentation, it is easier to produce upper bounds on the cost of memory allocation and on the amount of required memory. The total amount of memory required in order to guarantee that an allocation request can always be met will be smaller for a system with compacting garbage collection than for a system using non-compacting manual memory management or static memory management in the general case.
- The complexity of embedded systems keeps increasing as more and more functionality is implemented in software and more dynamic behaviour is required. Garbage collection is a scalable technique for handling the growing problem of memory management. Static memory management does not scale up as the size of the software grows, nor does dynamic manual memory management. Using automatic memory management, it will be possible to develop maintainable embedded systems quicker and to a lower cost.
- Object orientation is a software development technique becoming evermore popular and it has recently started to make its way into hard real-time computing. Dynamic memory management fits well into the dynamic execution model implied by object orientation. Automatic memory management will be required to handle the complexity of object orientation in a robust way. The growing interest for using the object-oriented language Java, which includes garbage collection, in embedded applications illustrates this need.

Altogether, automatic memory management is a vital technique that will be necessary when we are going to design and implement the real-time systems of tomorrow.

Bibliography

[AB91] L. Andersson, A. Blomdell. A Real-Time Programming Environment and a Real-Time Kernel. In *Proceedings of the National Swedish Symposium on Real-Time Systems*. 1991.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, 1991.

[ADVW92] J.L. Armstrong, B. O. Däcker, S. R. Virding, M. C. Williams. Implementing a Functional Language for Highly Parallel Real Time Applications. In *Proceedings of SETSS'92.* Florence, Italy, March, 1992.

[AEL88] A. W. Appel, J. R. Ellis, K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, 1988.

[ANSI78] *American National Standard Programming Language Fortran*, X3.9-1978, American National Standards Institute (ANSI), 1978.

[ATT88] *WE DSP32 and DSP32C Support Software Library – User Manual*, AT&T, 1988.

[Bad93] S. B. Baden. Low-Overhead Storage Reclamation in the Smalltalk-80 Virtual Machine. In G. Krasner, editor, *Smalltalk-80 – Bits of History, Words of Advice.* Addison-Wesley, Reading, Mass., 1993.

[Bak78] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, Vol. 21, No. 4, April, 1978.

[Bak92] H. G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices*, Vol. 27, No. 3, March 1992.

[Bar88] J. F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots*. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, California, February, 1988.

[BDS91] H-J. Boehm, A. J. Demers, S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*. Toronto, Canada, June, 1991.

[Ben90] M. Bengtsson. *Real-Time Garbage Collection*. Licentiate thesis, Dept. of Computer Science, Lund University, 1990.

[BMN+60] J. W. Backus, J. McCarthy, P. Naur, A. van Wijngaarden, & al. *Report on the Algorithmic Language ALGOL 60*. Regnecentralen, Copenhagen, 1960.

[Bob80] D. G. Bobrow. Managing Re-entrant Structures Using Reference Counts. *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, July, 1980.

[Bre89] R. P. Brent. Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, July, 1989.

[Bro84] R. A. Brooks. Trading Data Space for Reduced Time And Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings of the 1984 Symposium on Lisp and Functional Programming*. Austin, Texas, August, 1984.

[BS93] S. Ballard, S. Shirron. The Design and Implementation of VAX/Smalltalk-80. In G. Krasner, editor, *Smalltalk-80 – Bits of History, Words of Advice*. Addison-Wesley, Reading, Mass., 1993.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[BW88] H-J. Boehm, M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice & Experience*, Vol. 18, No. 9, September, 1988.

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. Communications of the ACM, Vol. 13, No. 11, November, 1970.

[Chr84] T. W. Christopher. Reference Count Garbage Collection. *Software Practice and Experience*, Vol. 14, No. 6, June, 1984.

[Coh81] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, Vol. 13, No. 3, September, 1981.

[Col60] G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, Vol. 3, No. 12, December, 1960.

[DB76] L. P. Deutsch, D. G. Bobrov. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, Vol. 19, No. 9, September, 1976.

[Der74] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. *Information Processing 74*, North-Holland Publishing Company, 1974.

[Dij68] E. W. Dijkstra. Cooperating Sequential Processes. In *Programming Languages*. F. Genuys (ed), Academic Press, 1968.

[DLM+78] E. W. Dijsktra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, Vol. 21, No. 11, November, 1978.

[Eke97] J. Eker. *A Framework for Dynamically Configurable Embedded Controllers*. Licentiate thesis, Dept. of Automatic Control, Lund Institute of Technology, 1997.

[ERS85] *PASCAL/D80 Users Instruction*. Ericsson Radio System AB, No. 198 17-LXA 105 01 Ue, February, 1985.

[EV91] S. L. Engelstad, J. E. Vandendorpe. *Automatic Storage Management for Systems With Real Time Constraints*, OOPSLA'91 Workshop: Garbage Collection in Object Oriented Systems, 1991.

[Fol93] D. E. Folkesson. *Principer för realtidsexekvering i svenska militära avioniksystem*. SNART'93, National Swedish Symposium on Real-Time Systems, August, 1993.

[FY69] R. Fenichel, J. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, Vol. 12, No. 11, November, 1969.

[HEM92] R. L. Hudson, J. Eliot, B. Moss. Incremental Collection of Mature Objects. In *Proceedings of IWMM'92*, Springer-Verlag, LNCS 637, St. Malo, France, 1992.

[Hen94] R. Henriksson. Scheduling Real Time Garbage Collection. In *Proceedings of NWPER'94*, Lund, Sweden, 1994.

[Hen96] R. Henriksson. *Scheduling Real-Time Garbage Collection*. Licentiate thesis, Dept. of Computer Science, Lund University, 1996.

[Hen97] R. Henriksson. *Predictable Automatic Memory Management for Embedded Systems*, OOPSLA'97 Workshop on Garbage Collection and Memory Management, Atlanta, Georgia, October, 1997.

[HH89] V. P. Holmes, D. L. Harris. A Designer's Perspective of the Hawk Multiprocessor Operating System Kernel. *ACM Operating System Review*, Vol. 23, No. 3, July, 1989.

[Hil92] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architecture*. Seattle, Washington, April, 1992.

[I+83] J. D. Ichbiah & al. *Reference Manual for the ADA Programming Languag*e. Castle House Publications Ltd, 1983.

[Ive98] A. Ive. Runtime Performance Evaluation of Embedded Software. To appear in *Proceedings of NWPER'98*, Ronneby, Sweden, August, 1998.

[Joh92] R. E. Johnson. Reducing the Latency of a Real-Time Garbage Collector. *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, March, 1992.

[JL96] R. Jones, R. Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.

[JP86] M. Joseph, P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, Vol. 29, No. 5, 1986.

[JW85] K. Jensen, N. Wirth. *Pascal: User Manual and Report*. Springer-Verlag, 1985.

[KM93] J. L. Knudsen, O. L. Madsen. Conceptual Framework. In *Object-Oriented Environments - The Mjølner Approach*, edited by J.L. Knudsen & al., Prentice-Hall International Ltd, 1993.

[Knu73] D. E. Knuth. *The Art of Computer Programming,* Vol 1. Addison-Wesley, Reading, Mass., 1973.

[KR78] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[Kro85] S. Krogdahl. Multiple Inheritance in Simula-like Languages. *BIT*, Vol. 25, No. 2, 1985.

[Lar77] R. G. Larson. Minimizing Garbage Collection as a Function of Region Size. *SIAM Journal on Computing*, Vol. 6, No. 4, December, 1977.

[LH83] H. Lieberman, C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, Vol. 26, No. 6, June, 1983.

[Lin92] R. D. Lins. Cyclic Reference Counting With Lazy Mark-Scan. *Information Processing Letters*, Vol. 44, No. 4, 1992.

[LL73] C. L. Lui, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, 1973.

[LR80] B. W. Lampson, D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, Vol. 23, No. 2, 1980.

[Mar80] C. D. Marlin. *Coroutines*. Springer-Verlag, 1980.

[MH95] B. Magnusson, R. Henriksson. Garbage Collection for Control Systems. In *Proceedings of IWMM'95*, Springer-Verlag, LNCS 986, Kinross, Scotland, September, 1995.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, Vol. 3, No. 4, April, 1960.

[Min63] M. L. Minsky. *A Lisp Garbage Collector Algorithm Using Serial Secondary Storage*. Memo 58 (rev.) Project Mac, M.I.T., Cambridge, Mass., December, 1963.

[MMN93] O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard. *Object-Oriented Programming in the Beta Language*. Addison-Wesley Publishing Company, Reading, Mass., 1993.

[Moo84] D. A. Moon. Garbage Collection in a Large Lisp System. In *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, 1984.

[NO93] S. Nettles, J. O'Toole. Real-Time Replication Garbage Collection. In *Proceedings of SIGPLAN'93 Conference on Programming Langauges Design and Implementation, ACM Sigplan Notices*, Vol. 28, No. 6, June, 1993.

[Nil96] K. Nilsson. *Industrial Robot Programming*. Ph.D. thesis, Dept. of Automatic Control, Lund Institute of Technology, 1996.

[NS94] K. D. Nilsen, W. J. Schmidt. A High-Performance Hardware Assisted Real-Time Garbage Collection System. *Journal of Programming Languages*, Vol. 2, No. 1, March, 1994.

[Rob71] J. M. Robson. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *Journal of the ACM*, Vol. 18, No. 3, July, 1971.

[SG95] J. Seligmann, S. Grarup. Incremental Mature Garbage Collection Using the Train Algorithm. In *Proceedings of ECOOP'95*, Aarhus, Denmark, August, 1995.

[SIS87] *Data processing - Programming languages - SIMULA*. Swedish standard SS 636114. SIS, Stockholm, Sweden, 1987.

[SR89] J. A. Stankovic, K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *ACM Operating System Review*, Vol. 23, No. 3, July, 1989.

[SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*. Vol. 39, No. 9, 1990.

[SRL94] L. Sha, R. Rajkumar, J. P. Lehoczky. Generalized Rate-Monotonic Scheduling Theory. *Proceedings of the IEEE*. Vol. 82, No. 1, 1994.

[Ste75] G. L. Steele Jr. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM*, Vol. 18, No. 9, September, 1975.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.

[SW67] H. Schorr, W. M. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Communications of the ACM*, Vol. 10, No. 8, August, 1967.

[Tho76] L-E. Thorelli. A Fast Compactifying Garbage Collector, *BIT*, Vol. 16, No. 4, 1976.

[UJ88] D. Ungar, F. Jackson. Tenuring Policies for Generation-Based Storage Reclamation. In Proceedings of OOPSLA'88, *ACM SIGPLAN Notices*, Vol. 23, No. 11, November, 1988.

[Ung84] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, Vol. 19, No. 6, September, 1976.

[Vir95] R. Virding. A Garbage Collector for the Concurrent Real-Time Language Erlang. In *Proceedings of IWMM'95*, Springer-Verlag, LNCS 986, Kinross, Scotland, Septemeber, 1995.

[Wad76] P. L. Wadler. Analysis of an Algorithm for Real Time Garbage Collection. *Communications of the ACM*, Vol. 19, No. 9, September, 1976.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of IWMM'92*, Springer-Verlag, LNCS 637, St. Malo, France, September 1992.

[Wit92] P. T. Withington. How Real is "Real-Time" GC?, *OOPS Messenger*, October, 1992. OOPSLA'91 Workshop: Garbage Collection in Object Oriented Systems.

[WJ93] P. R. Wilson, M. S. Johnstone. *Real-Time Non-Copying Garbage Collection*. OOPSLA'93 Workshop on Memory Management and Garbage Collection, Washington D.C., ftp://ftp.cs.utexas.edu/pub/garbage/GC93/wilson.ps, October, 1993.

[WLM92] P. R. Wilson. Caching Considerations for Generational Garbage Collection. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California, ftp://ftp.cs.utexas.edu/pub/garbage/cache.ps, June, 1992.

[WRS95] *VxWorks Programmers Guide: version 5.3*, Wind River Systems, Alameda, California, 1995.

[Yua90] T. Yuasa. Real-Time Garbage Collection on General-Purpose Machines. *Journal of Systems and Software*, Vol. 11, No. 3, March, 1990.

[Zor89] B. G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, December, 1989.

[ÅW84] K. J. Åström, B. Wittenmark. Computer Controlled Systems – Theory and Design. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

Index

A
Ada 20
Algol 21
allocation
    high-priority process 52, 124
    low-priority process 53, 126
API 106
the Appel-Ellis-Li collector 137
application program interface.
        See API
atomic operation 60, 89
automatic control 9

B
Baker's algorithm 46, 135
Bartlett's collector 33
basic inheritance protocol 17
batch system 8
Beta 34, 100, 114
black object 46
blocking 15
Brook's algorithm 46, 136

C
C 21, 32, 99
C++ 32, 99, 112
cache 11, 39
Cheney's scanning strategy 28
coding convention 106
collector 23
compaction 23, 25, 44
concurrent garbage collection 39, 141
conservative algorithm 32
control delay 12
control response time 12
copying algorithm 23, 28, 29
coroutine 48, 99
correctness 1, 7

D
dangling pointer 3, 22
deadline 1, 7
deadline monotonic scheduling 16
degradation 65
`dispose` 22

E
earliest deadline first. *See* EDF
EDF 18
efficiency 34
Eiffel 114
embedded system 1, 9
Erlang 36, 136
evacuation 28
execution time 10
    worst-case. *See* WCET

F
first-fit 128
flip 29, 47
floating object 36
Fortran 21
forwarding pointer 29, 47, 107
fragmentation 3, 22, 44

`free` 22, 128
free-list 22, 23
fromspace 29, 47

G
garbage collection 4, 22
  scheduling 5
garbage collector 4
GC ratio 92
  current 58
  minimum 57
GC work metric 68
GC. *See* garbage collection
generation 33
generation-based algorithm 33, 93
graceful degradation 66
grey object 46

H
hardware-assisted GC 139
Hawk 19
heap 3, 101
high-priority GC process 52, 56, 104

I
immediate inheritance protocol 18
implementation 117
incrementallity 35
industrial robot 118
interactive system 8, 37
interrupt 61

J
JAS 39 20
Java 99, 114
jitter 16, 130

L
latency 12, 44, 64
lazy evacuation 53
light-weight process 7
Lisp 22
the LISP 2 algorithm 25, 26
logic analyser 118

M
`malloc` 22, 128
mark-compact 25
mark-sweep 23, 25, 26
memory compaction 3
memory fragmentation.
    See fragmentation
memory hierarchy 39
memory leak 3, 22
memory management
  automatic 3, 22
  dynamic 2, 3, 21
  manual 3, 21, 101
  static 2, 21
memory pool 2
metric. *See* GC work metric
Modula-2 99
multiple inheritance 112
mutation log 61
mutator 23

N
`new` 22

O
object initialization 51
object layout information 107
object orientation 2
operating system process 7
overhead 4, 62
overload 15, 19, 65

P
Pascal 21
Pascal/D80 2
performance
  average-case 10
  worst-case 10
predictability 2, 10, 73
preemption 15
priority ceiling protocol 18

priority inheritance protocol 16
priority inversion 16, 17
process 7
  high-priority 9
  low-priority 9
process scheduler 13
process stack 101
processor utilization 15, 19
Pålsjö 121

Q
QNX 19

R
rate monotonic analysis. *See* RMA
rate monotonic scheduling. *See* RMS
read barrier 35, 50, 89
real-time
  demand 7
  kernel 19, 98
  requirement 7
  system 1, 7
    hard 9
    soft 9
reference counting 23
replicating garbage collection 137
response time 4, 12
RMA 15
RMS 15, 73
root pointer 25, 29, 48, 102

S
sample delay 12
sampling 9, 11
scan pointer 28, 48
scanning 29
schedulability analysis 10
scheduling
  dynamic 14
  earliest deadline first. *See* EDF
  fixed-priority 14
  static 13
scheduling analysis 73
scheduling strategy 38, 43
semi-concurrent GC 46
semi-concurrent scheduling 44
semispace 28
sequential garbage collection 38
service time 63
Simula 22, 99, 100, 114
Spring 20
stop-the-world 36
stop-the-world algorithm 34
synchronization 60

T
task 1, 7
tenuring 33
test application 120
Thorelli's algorithm 26
thread 7
tospace 29, 47
tracing algorithm 25
the train algorithm 34, 37
the Treadmill 138
tri-colour marking 46

V
VME board 98
VxWorks 19

W
WCET 11
white object 46
write barrier 35, 50, 89

Y
Yuasa's algorithm 138